# ByteTransformer: A High-Performance Transformer Boosted for Variable-Length Inputs

Yujia Zhai,*¶ Chengquan Jiang,†¶ Leyuan Wang,† Xiaoying Jia,† Shang Zhang,‡
Zizhong Chen,* Xin Liu,†§ Yibo Zhu†
*University of California, Riverside
†ByteDance Ltd.
‡NVIDIA Corporation
§Correspondence to liuxin.ai@bytedance.com
¶These authors contributed equally to this work.

arXiv:2210.03052v4 [cs.LG] 20 Feb 2023

***Abstract*—Transformers have become keystone models in natural language processing over the past decade. They have achieved great popularity in deep learning applications, but the increasing sizes of the parameter spaces required by transformer models generate a commensurate need to accelerate performance. Natural language processing problems are also routinely faced with variable-length sequences, as word counts commonly vary among sentences. Existing deep learning frameworks pad variable-length sequences to a maximal length, which adds significant memory and computational overhead. In this paper, we present ByteTransformer, a high-performance transformer boosted for variable-length inputs. We propose a padding-free algorithm that liberates the entire transformer from redundant computations on zero padded tokens. In addition to algorithmic-level optimization, we provide architecture-aware optimizations for transformer functional modules, especially the performance-critical algorithm Multi-Head Attention (MHA). Experimental results on an NVIDIA A100 GPU with variable-length sequence inputs validate that our fused MHA outperforms PyTorch by 6.13x. The end-to-end performance of ByteTransformer for a forward BERT transformer surpasses state-of-the-art transformer frameworks, such as PyTorch JIT, TensorFlow XLA, Tencent TurboTransformer, Microsoft DeepSpeed-Inference and NVIDIA FasterTransformer, by 87%, 131%, 138%, 74% and 55%, respectively. We also demonstrate the general applicability of our optimization methods to other BERT-like models, including ALBERT, DistilBERT, and DeBERTa.**

***Index Terms*—Transformer, BERT, Multi-head Attention, MHA, Natural Language Processing, NVIDIA GPU, CUTLASS**

## I. Introduction

The transformer model [1]is a proven effective architecture widely used in a variety of deep learning (DL) applications, such as language modeling [2], [3], neural machine translation [1], [4] and recommendation systems [5], [6]. The last decade has witnessed rapid developments in natural language processing (NLP) pre-training models based on the transformer model, such as Seq2seq [1], GPT-2 [7] and XLNET [3], which have also greatly accelerated the progress of NLP. Of all the pre-training models based on transformers, Bidirectional Encoder Representations from Transformers (BERT), proposed in 2018 [2], is arguably the most seminal, inspiring a series of subsequent works and outperforming reference models on a dozen NLP tasks at the time of creation.

BERT-like models consume increasingly larger parameter space and correspondingly more computational resources. When BERT was discovered, a large model required 340 million parameters [8], but currently a full GPT-3 model requires 170 billion parameters [9]. The base BERT model requires 6.9 billion floating-point operations to inference a 40-word sentence, and this number increases to 20 billion when translating a 20-word sentence using a base Seq2Seq model [10]. The size of the parameter space and the computational demands increase the cost of the training and inference for BERT-like models, which requires the attention of the DL community in order to accelerate these models.

To exploit hardware efficiency, DL frameworks adopt a batching strategy, where multiple batches are executed concurrently. Since batched execution requires task shapes in different batches to be identical, DL frameworks presume fixed-length inputs when designing the software [11]–[14]. However, this assumption cannot always hold, because transformer models are often faced with variable-length input problems [8], [10]. In order to deploy models with variable-length inputs directly to conventional frameworks that support only fixed-length models, a straightforward solution is to pad all sequences with zeros to the maximal sequence length. However, this immediately brings in redundant computations on wasted padded tokens. These padded zeros also introduce significant memory overhead that can hinder a large transformer model from being efficiently deployed.

Existing popular DL frameworks, such as Google TensorFlow with XLA [15], [16], Meta PyTorch with JIT [17], and OctoML TVM [18], leverage the domain-specific just-in-time compilation technique to boost performance. Another widely-adopted strategy to generate low-level performance optimization is delicate manual tuning: NVIDIA TensorRT [19], a DL runtime, falls into this category. Yet all of these frameworks require the input sequence lengths to be identical to exploit the speedup of batch processing. To lift the restriction on fixed sequence lengths, Tencent [10] and Baidu [8] provide explicit support for models with variable sequence lengths. They group sequences with similar lengths before launching

We have made ByteTransformer open-source and available at a public GitHub repository: https://github.com/bytedance/ByteTransformer.

batched kernels to minimize the padding overhead. However, this proactive grouping approach still introduces irremovable padding overhead when grouping and padding sequences with similar yet different lengths.

In contrast to training processes that can be computed offline, the inference stage of a serving system must be processed online with low latency, which imposes high performance requirements on DL frameworks. A highly efficient DL inference framework for NLP models requires delicate kernel-level optimizations and explicit end-to-end designs to avoid wasted computations on zero tokens when handling variable-length inputs. However, existing DL frameworks do not meet these expectations. In order to remedy this deficit, we present ByteTransformer, a highly efficient transformer framework optimized for variable-length inputs in NLP problems. We not only design an algorithm that frees the entire transformer of padding when dealing with variable-length sequences, but also provide a set of hand-tuned fused GPU kernels to minimize the cost of accessing GPU global memory. More specifically, our contributions include:

- We design and develop ByteTransformer, a high-performance GPU-accelerated transformer optimized for variable-length inputs. ByteTransformer has been deployed to serve world-class applications including TikTok and Douyin of ByteDance.
- We propose a padding-free algorithm that packs the input tensor with variable-length sequences and calculates the positioning offset vector for all transformer operations to index, which keeps the whole transformer pipeline free from padding and calculations on zero tokens.
- We propose a fused Multi-Head Attention (MHA) to alleviate the memory overhead of the intermediate matrix, which is quadratic to the sequence length, in MHA without introducing redundant calculations due to padding for variable-length inputs. Part of our fused MHA has been deployed in the production code base of NVIDIA CUTLASS.
- We hand-tune the memory footprints of layer normalization, adding bias and activation to squeeze the final performance of the system.
- We benchmark the performance of ByteTransformer on an NVIDIA A100 GPU for forward pass of BERT-like transformers, including BERT, ALBERT, DistilBERT, and DeBERTa. Experimental results demonstrate our fused MHA outperforms standard PyTorch attention by 6.13X. Regarding the end-to-end performance of standard BERT transformer, ByteTransformer surpasses PyTorch, TensorFlow, Tencent TurboTransformer, Microsoft DeepSpeed and NVIDIA FasterTransformer by 87%, 131%, 138%, 74%, and 55%, respectively.

The rest of the paper is organized as follows: we introduce background and related works in Section II, and then detail our systematic optimization approach in Section III. Evaluation results are given in Section IV. We conclude our paper and present future work in Section V.

## II. Background and Related Works

We provide an overview of the transformer model, including its encoder-decoder architecture and multi-head attention layer. We also survey related works on DL framework acceleration.

### A. The transformer architecture

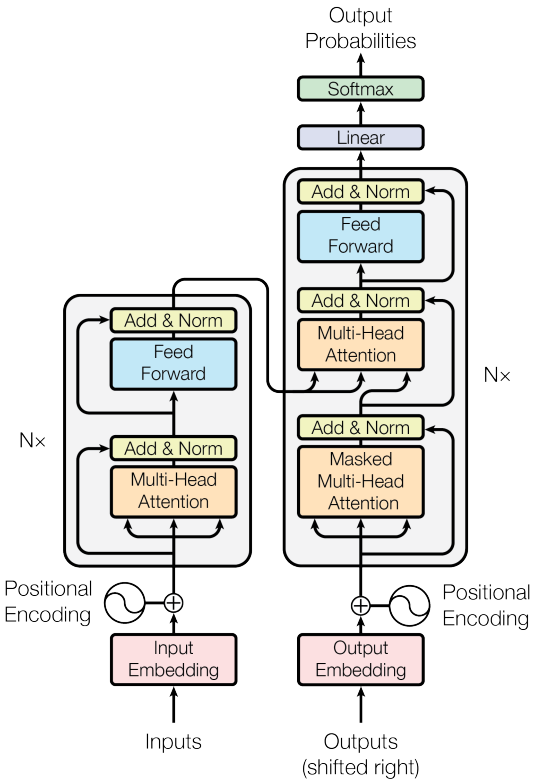



Fig. 1: The transformer architecture. [1]

Figure 1 shows the encoder-decoder model architecture of the transformer. It consists of stacks of multiple encoder and decoder layers. In an encoder layer, there is a multi-head attention layer followed by a feed-forward network (FFN) layer. A layer normalization (layernorm) operation is applied after both MHA and FFN. In a decoder layer, there are two sets of consecutive MHA layers and one FFN layer, and each operation is normalized with a layernorm. The FFN is used to improve the capacity of the model. In practice, FFN is implemented by multiplying the tensor by a larger scaled tensor using GEMM. Here we skip the embedding descriptions in the figure, and refer an interested reader to [1] for details. Although we show both encoder and decoder modules for this transformer, a BERT transformer model only contains the encoder section [2]. In this paper, we present optimizations for BERT-like transformer models, which can be extended to other transformers containing decoder sections.

Self-attention is a key module of the transformer architecture. Conceptually, self-attention computes the significance of each position of the input sequence, with the information from other positions considered. A self-attention receives three input tensors: query (Q), key (K), and value (V). Self-attention can

be split into multiple heads. The Q and K tensors are first multiplied ($1^{st}$ GEMM) to compute the dot product of the query against all keys. This dot product is then scaled by the hidden dimension $d_k$ and passed through a softmax function to calculate the weights corresponding to the value tensor. Each head of the output tensor is concatenated before going through another linear layer by multiplying against tensor V ($2^{nd}$ GEMM). Expressing self-attention as a mathematical formula, we have:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}}) \times V \quad (1)$$

Whereas the formula of multi-head attention is: $Multihead(Q, K, V) = Concat(head_i, ..., head_h)$, here $head_i = Attention(Q_i, K_i, V_i)$.

### B. Related works on DL acceleration

Performance is a crucial aspect in the real-world deployment of software systems, attracting significant attention across various applications [20]–[22], including DL frameworks. The conventional DL frameworks, such as PyTorch, TensorFlow, TVM, and TensorRT are designed explicitly for fixed-length input tensors. When dealing with NLP problems with variable-length input, all sequences are padded to the maximal length, which leads to significant wasted calculations on zero tokens. A few DL frameworks, such as Tencent TurboTransformer [10] and NVIDIA FasterTransformer [23], employ explicit designs for variable-length inputs. TurboTransformer designs run-time algorithms to group and pad sequences with similar lengths to minimize the padding overhead. TurboTransformer also uses a run-time memory scheduling strategy to improve end-to-end performance. Kernel-level optimizations are of the same significance as algorithmic optimizations. NVIDIA's FasterTransformer uses vendor-specific libraries such as TensorRT and cuBLAS [24] as its back-end, which provide optimized implementations of various operations at the kernel level.

Other end-to-end DL frameworks have also presented optimizations for BERT-like transformers, such as E.T. [25] and DeepSpeed-Inference [26]. E.T. introduces a novel MHA architecture for NVIDIA Volta GPUs and includes pruning designs for end-to-end transformer models. In contrast, ByteTransformer targets unpruned models and is optimized for NVIDIA Ampere GPUs. DeepSpeed-Inference is optimized for large distributed models on multiple GPUs, while ByteTransformer currently focuses on lighter single-GPU models.

In addition to end-to-end performance acceleration, the research community has also made focused efforts to improve a key algorithm of the transformer, multi-head attention. PyTorch provides a standard implementation of MHA [27]. NVIDIA TensorRT utilizes a fused MHA for short sequences with lengths up to 512, as described in [28]. To handle longer sequences, FlashAttention was proposed by Stanford researchers in [29]. FlashAttention assigns the workload of a whole attention unit to a single threadblock (CTA). However, this approach can result in underutilization on wide GPUs when there are not enough attention units assigned. Our fused MHA, on the other hand, provides high performance for both short and long sequences for variable-length inputs without leading to performance degradation in small-batch scenarios.

TABLE I. Summarizing state-of-the-art transformers.

| | variable-len support | kernel tuning | fused MHA | kernel fusion |
|---|---|---|---|---|
| Tensorflow XLA | no | yes | no | no |
| PyTorch JIT | no | yes | no | no |
| FasterTransformer | yes | yes | $\leq 512$ | no |
| TurboTransformer | yes | yes | no | partially |
| **ByteTransformer** | **yes** | **yes** | **yes** | **yes** |

Table I surveys state-of-the-art transformers. TensorFlow and PyTorch provide tuned kernels but require padding for variable-length inputs. NVIDIA FasterTransformer and Tencent TurboTransformer, although providing support for variable-length inputs, do not perform comprehensive kernel fusion or explicit optimization for the hot-spot algorithm MHA for any length of sequence. In addition, TurboTransformer only optimizes part of the fusible operations in the transformer model, such as layernorm and activation, namely 'partial kernel fusion' in the table. Our ByteTransformer, in contrast, starting with a systemic profiling to locate bottleneck algorithms, precisely tunes a series of kernels including the key algorithm MHA. We also propose a padding-free algorithm which completely removes redundant calculations for variable-length inputs from the entire transformer.

## III. Designs and Optimizations

In this section, we present our algorithmic and kernel-level optimizations to improve the end-to-end performance of BERT transformer under variable-length inputs.

### A. Math expression of BERT transformer encoder

Figure 2(a) illustrates the architecture of the transformer encoder. The input tensor is first processed through the BERT pipeline, where it is multiplied by a built-in attribute matrix to perform Q, K, and V positioning encoding. This operation can be implemented using three separate GEMM operations or in batch mode. Realizing that the corresponding attribute matrices to Q, K, and V are all the same shape (hidden_dim x hidden_dim), we pack them to continuous memory space and launch a single batched GEMM kernel that calculates Q, K, and V to reduce the kernel launch overhead at runtime. Bias matrices for Q, K, and V are then added to the encoded tensor, which is passed through the self-attention module. In addition to the multi-head attention module, the BERT transformer encoder includes projection, feed forward network, and layer normalization. The encoder pipeline can be represented as a series of mathematical operations, including six GEMMs (shown in light purple) and other memory-bound operations (shown in light blue).

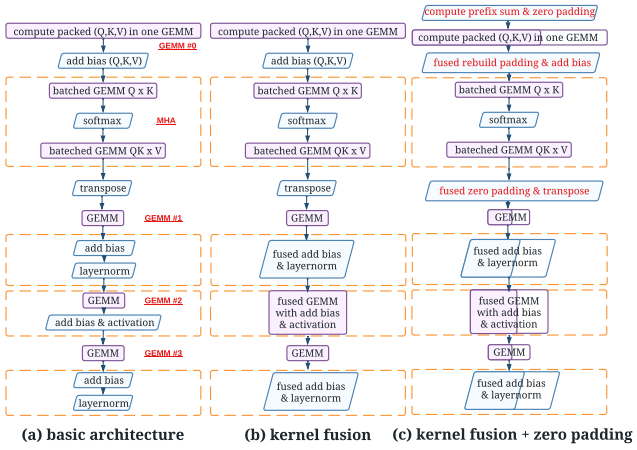
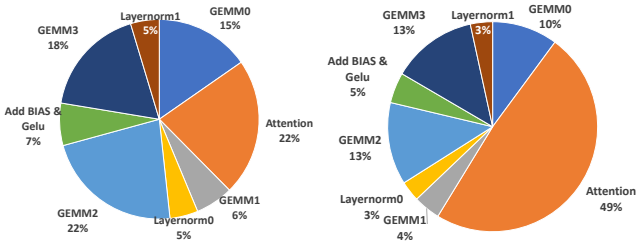



Fig. 2: BERT transformer architecture and optimizations.

### B. Profiling for single-layer standard BERT transformer

We implement the pipeline of Figure 2 (a) by calling cuBLAS and profile its single-layer performance on an NVIDIA A100 GPU. We adopt the standard BERT transformer configuration (batch size: 16, head number: 12, head size: 64) and profile for two different sequence lengths: 256 and 1024.

Figure 3 shows the performance breakdown for two sequence lengths. `GEMM0` to `GEMM3` refer to the consecutive four GEMMs that are enumerated from GEMM #0 to GEMM #3 in Figure 2 (a). The other two batched GEMMs are part of the attention module and are therefore profiled together with the softmax as a whole, referred to as `MHA` in Figure 3. The two sets of "add bias and layernorm" operations are referred to as `layernorm0` and `layernorm1`. The profiling results show that the compute-bound GEMM operations account for 61% and 40% of the total execution time for both test cases. The attention module, which includes a softmax and two batched GEMMs, is the most time-consuming part of the transformer. As the sequence length increases to that of a GPT-2 model (1024), attention accounts for 49% of the total execution time, while the remaining memory-bound operations (layernorm, add bias and activation) only take up 11%-17%.




(a) Sequence lengths 256 (b) Sequence lengths 1024

Fig. 3: Performance breakdown of forward BERT transformer.

### C. Fusing memory-bound operations of BERT transformer

Since cuBLAS uses architectural-aware optimizations for high performance GEMMs, presumably there remain limited opportunities for further acceleration. Therefore, we turn our eyes to optimizing the modules containing memory-bound operations, such as attention (with softmax), feed forward network (with layernorm) and add bias followed by element-wise activation. We improve these operations by fusing distinct kernels and reusing data in registers to reduce global memory access. Figure 2 (b) presents the BERT transformer pipeline with memory-bound kernel fusion, where we fuse layernorm and activation with their consecutive kernels.

*1) Add bias and layer normalization:* These operations account for 10% and 6% of the overall execution time for sequence lengths 256 and 1024, respectively. After MHA, the result tensor (`valid_word_cnt` $\times$ `hidden_dim`) needs to first be added upon the input tensor (bias) and perform layer normalization. Here hidden dimension (`hidden_dim`) equals `head_num` $\times$ `head_size`. In standard BERT configuration, head number and head size are fixed to 12 and 64. The naive implementation introduces two rounds of memory access to load and store the tensor. We provide a fused kernel that only needs to access the global memory in one round to finish both layernorm and adding bias. Kernel fusion for this sub-kernel improves the performance by 61%, which accordingly increases the single-layer BERT transformer performance by 3.2% for sequence lengths ranging 128 to 1024 in average.

*2) add bias and activation:* These operations account for 7% and 5% of the overall execution time for sequence lengths 256 and 1024, respectively. After the projection via matrix multiplication, the result tensor will be added against the input tensor and perform an element-wise activation using GELU [30]. Our fused implementation, rather than storing the GEMM output to global memory and loading it again to conduct adding bias and activation, re-uses the GEMM result matrix at the register level by implementing a customized and fused CUTLASS [31] epilogue. Experimental results validate that our fused GEMM perfectly hides the memory latency of bias and GELU into GEMM. After this step, we further improve the single-layer BERT transformer by 3.8%.

### D. The zero padding algorithm for variable-length inputs

Because the real-time serving process receives sentences with various words as input tensor, the sequence lengths can often be different among batches. For such an input tensor composed of sentences with variable lengths, the conventional solution is to pad them to the maximal sequence length with useless tokens, which leads to significant computational and memory overhead. In order to address this issue, we propose the zero padding algorithm to pack the input tensor and store the positioning information for other transformer operations to index the original sequences.

Figure 4 presents the details of the zero padding algorithm. We use an input tensor with 3 sentences (proceeded in 3 batches) as an example. The longest sentence contains 5 word tokens while the other two have 2 and 4 words. The height

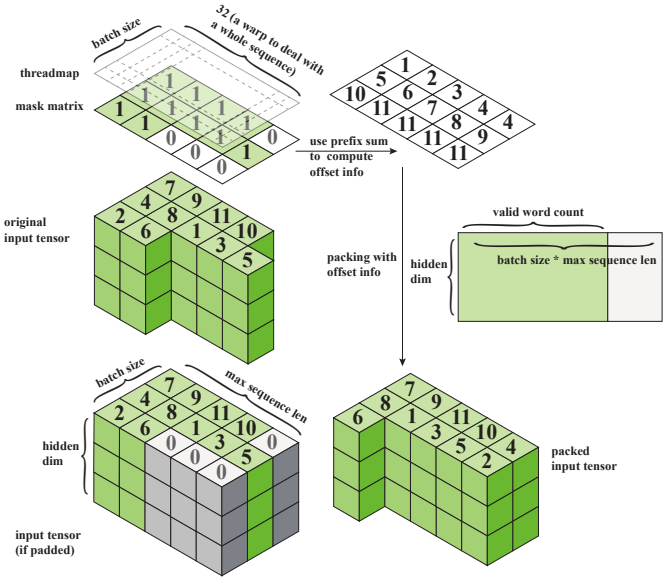



Fig. 4: The zero padding algorithm.

of the sample input tensor is 3, which is equal to the hidden dimension. The conventional method is to pad all sentences to the maximal sequence length by filling zeros. The elements, either 1 or 0, of the mask matrix correspond respectively to a valid token or a padded token of an input tensor with variable size. By calculating the prefix sum of the mask matrix, we can skip the padded tokens and provide the position indices of all valid tokens. We implement an efficient CUDA kernel to calculate the prefix sum and the position offset. Each warp computes the prefix sum for tokens of a whole sentence, so in total there are `batch_size` warps assigned in each threadblock for prefix sum calculation. Once the prefix sum is computed, we pack the input tensor to a continuous memory area so that the total number of words used in future calculations is reduced from `seq_len` $\times$ `batch_size` to the actual valid word count of the packed tensor.

Figure 2 (c) presents the detailed modifications on BERT by introducing our zero padding algorithm. Before conducting the positioning encoding, we calculate the prefix sum of the mask matrix to pack the input tensor so that we avoid computations on useless tokens in the first GEMM. Since batched GEMM in MHA requires identical problem shapes among different batches, we unpack the tensor before entering the attention module. Once MHA is completed, we pack the tensor again such that all remaining operations can benefit from the zero padding algorithm. The final result tensors are validated element-by-element against TensorFlow such that the correctness and accuracy are ensured. It is worth mentioning that padding and remove padding operations are fused with existing memory-bound footprints such as adding bias and transpose to minimize the overhead led by this feature.

Our presented padding-free algorithm is designed to ensure semantic preservation. We maintain an array that stores the mapping relationship of the valid tokens between the original tensor and the packed tensor. The transformer operates on the packed tensor, and intermediate operations, such as MHA, layernorm and activation, refer to this position array to ensure the correctness. At the end of each layer, we reconstruct the output tensor according to the position array such that the whole pipeline is semantic preserving.

| | Baseline | Zero Padding | Zero Padding + fused MHA |
|---|---|---|---|
| GEMM0 | $6mk^2$ | $6(\alpha \cdot m)k^2$ | $6(\alpha \cdot m)k^2$ |
| MHA | $4\frac{m^2}{bs}k$ | $4\frac{m^2}{bs}k$ | $4\frac{(\alpha \cdot m)^2}{bs}k$ |
| GEMM1 | $2mk^2$ | $2(\alpha \cdot m)k^2$ | $2(\alpha \cdot m)k^2$ |
| GEMM2 | $8mk^2$ | $8(\alpha \cdot m)k^2$ | $8(\alpha \cdot m)k^2$ |
| GEMM3 | $8mk^2$ | $8(\alpha \cdot m)k^2$ | $8(\alpha \cdot m)k^2$ |

TABLE II. The computation number needed for variable-length inputs, where average sequence length = $\alpha$ * maximum, $m$ denotes `batch_size` $\cdot$ `max_seq_len`, $k$ is denote hidden dimension `head_num` $\cdot$ `head_size`, $bs$ denotes the batch size.

Table II counts the floating point computations of a single-layer BERT transformer. The computations of memory-bound operations are not included since they are negligible compared with the listed modules. Enabling the zero padding algorithm eliminates redundant computations for all compute-intensive modules other than MHA due to the restrictions of batched GEMM. When the average sequence length is equal to 60% of the maximum, turning on the zero padding algorithm further accelerates the BERT transformer by 24.7%.

### *E. Optimizing multi-head attention*

The zero-padding algorithm, although it effectively reduces wasted calculations for variable-length inputs, cannot directly benefit batched GEMM operations in MHA. This disadvantage becomes increasingly significant when the sequence length increases, as demonstrated in Table II. The complexity of MHA is quadratic to the sequence length, while the complexity of all other GEMMs is linear to the sequence length. This motivates us to provide a high-performance fused MHA while maintaining the benefits of the zero-padding algorithm. With our fused MHA, attention no longer faces redundant calculations on useless tokens, as shown in Table II.

*1) Unpadded fused MHA for short sequences:* For short input sequences, we hold the intermediate matrix in shared memory and registers throughout the MHA computation kernel to fully eliminate the quadratic memory overhead. We also access Q, K, and V tensors according to the positioning information obtained in the prefix sum calculation step to avoid redundant calculations on padding zeros for the MHA module.

Algorithm III.1 shows the pseudo code of our fused MHA for short sequences. We launch a 3-dimensional grid map: {`head_num`, `seq_len`/`split_seq_len`, `batch_size`}. Here `split_seq_len` is a user-defined parameter to determine the size of a sequence tile preceded by a threadblock (typically set to 32 or 48). The warp count of a threadblock is computed by the maximal sequence length: `split_seq_len`$/16 \times$ (`seq_len`$/16$). Each threadblock loads a chunk of $Q$ (`split_seq_len` $\times$ `head_size`), $K$ (`max_seq_len` $\times$ `head_size`) and $V$ ((`head_size` $\times$

**Algorithm III.1:** Unpadded fused MHA for short sequences

```
/* define skew offset to avoid bank conflict */
#define SKEW_HALF 8
Shared memory:
__half s_kv [max_seq_len][size_per_head + SKEW_HALF];
__half s_query [split_seq_len][size_per_head +
  SKEW_HALF];
__half s_logits [max_seq_len][size_per_head +
  SKEW_HALF];
/* warps collaboratively fill s_query with adding bias fused */
Load __half2 q_bias
for seq_id = warp_id : warp_num : split_seq_len do
    query = Q[batch_seq_offset + seq_id +
      thread_offset];
    offset = seq_id*(head_size+SKEW_HALF)+(lane_id*2);
    (__half2 *)s_query[offset] = fast_add(query,
      k_bias);
/* warps collaboratively fill s_kv with adding bias fused */
Load __half2 k_bias
for seq_id = warp_id : warp_num : batch_seq_len do
    key = K[batch_seq_offset + seq_id +
      thread_offset];
    offset = seq_id*(head_size+SKEW_HALF)+(lane_id*2);
    (__half2 *)s_kv[offset] = fast_add(key, k_bias);
/* compute Q*K using WMMA */
Clear wmma fragment QK to zero
for k_id = 0 : head_size / 16 do
    Load 16x16 wmma fragments of Q
    Load 16x16 wmma fragments of K
    Update QK = Q * K + QK using wmma::mma_sync
Store fragment QK to s_logits using wmma::store_matrix_sync
/* Compute softmax */
for seq_id = warp_id : warp_num : batch_seq_len do
    float logits[max_seq_len];
    each thread loads a whole sequence to fill local registers
    /* 1st round of reduction with register-level data re-use*/
    compute max_val in local registers
    /* register-level data re-use*/
    compute P = exp(P − max) and update local registers
    /* 2st round of reduction with register-level data re-use*/
    compute sum_val in local registers
    /* register-level data re-use*/
    compute P = P/sum_val and stream to s_logits
/* warps collaboratively fill s_kv with adding bias fused */
Load __half2 v_bias
for seq_id = warp_id : warp_num : batch_seq_len do
    value = V[batch_seq_offset + seq_id +
      thread_offset];
    offset = seq_id*(head_size+SKEW_HALF)+(lane_id*2);
    (__half2 *)s_kv[offset] = fast_add(value, v_bias);
/* Similar to Q * K so omitting the details here */
Compute P * V using wmma and stream to global memory
```

max_seq_len)) into shared memory and computes MHA for a tile of the result tensor. We allocate three shared-memory buffers to hold $Q$, $K$, $V$ sub-matrices. Due to the algorithmic nature of MHA, we can re-use $K$ and $V$ chunks in the same shared-memory buffer `s_kv`. The intermediate matrix of MHA is held and re-used in another pre-allocated shared-memory buffer `s_logits`.

The workflow of fused MHA for short sequences is straightforward yet efficient. Each thread first loads its own tile of $Q$ and $K$ into shared memory and computes GEMM for $P = Q \times K$. The element-wise adding bias and scaling operations are both fused with the load process to hide the memory latency. GEMM is computed using the CUDA `wmma` intrinsic to leverage tensor cores of NVIDIA Ampere GPUs. The intermediate matrix $P$ is held in shared memory during

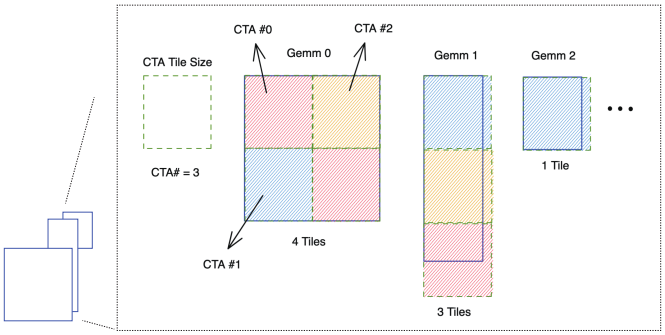



Fig. 5: Grouped GEMM demonstration.

the reduction. Because we explicitly design this algorithm for short sequences, each thread can load a whole sequence of $P$ from shared memory into register files for both reduction and element-wise exponential transform in softmax. Once the softmax operation is completed, we load a $K$ tile to shared memory to compute the second GEMM $O = P \times V$, and then store the result tensor $O$ to the global memory.

*2) Unpadded fused MHA for long sequences:* Because of the limited resources of register files and shared memory, the previous fused MHA is no longer feasible for long sequences. Therefore, we set 384 to be the cut-off sequence length and propose a grouped GEMM based fused MHA for large models.

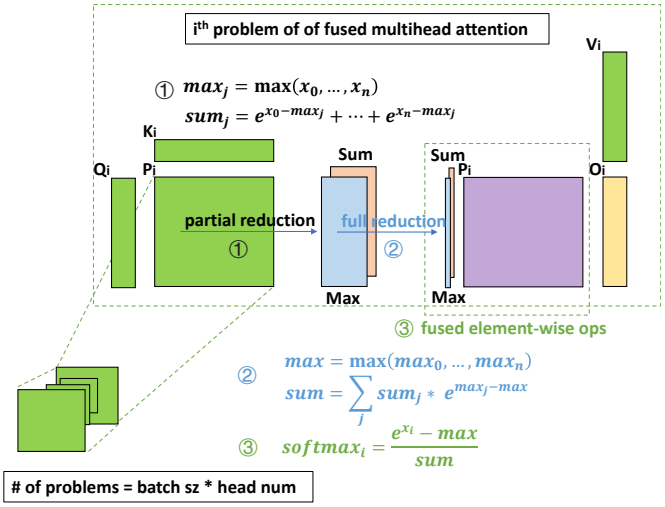



Fig. 6: Grouped-GEMM-based FMHA. The prototype of our fused MHA has been upstreamed to and released with CUTLASS 2.10. Source codes are available at [32].

The Grouped GEMM idea is first presented by NVIDIA CUTLASS [31]. Different from batched GEMM, where all GEMM sub-problems are required to have an identical shape, grouped GEMM allows arbitrary shapes for sub-problems. This is enabled by a built-in scheduler that iterates over all GEMM sub-problems in a round-robin manner. Figure 5 demonstrates the idea of grouped GEMM using an example with 3 sub-problems. Supposing 3 threadblocks (CTAs) are launched, each CTA calculates a fix-sized CTA tile at each

step until all GEMM sub-problems have been covered. GPU computes in waves, logically. In the first wave, All three CTAs calculate 3 tiles (light red, light yellow and light blue in the figure). And then in the second CTA wave, CTA #0 moves to the bottom-right tile of GEMM 0 while CTA #1 and CTA #2 move to sub-problems of GEMM 1. In the final CTA wave, CTA #0 and CTA #1 continue to compute tasks in GEMM 1 and GEMM 2 while CTA #2 keeps idle because there are no more available tiles in the computational graph.

Since grouped GEMM lifts the restriction on the shape of sub-problems, it can directly benefit MHA problems with variable-length inputs. Figure 6 presents our grouped-GEMM-based fused MHA for long sequences. The total number of MHA problems is equal to `batch_size` $\times$ `head_num`. The MHA problems among different batches have different sequence lengths, while sequence lengths within the same batch are identical. The grouped GEMM scheduler iterates over all attention units in a round-robin manner. In each attention unit, we first compute GEMM $P_i = Q_i \times K_i$, and conduct softmax on $P_i$. The second GEMM $O_i = P_i \times V_i$ provides us with the final attention result. Here $i$ indicates the $i^{th}$ problem of grouped MHA with variable shapes. The softmax operation is fused with GEMMs to hide the memory latency. We have upstreamed the prototype of our grouped GEMM based fused MHA into NVIDIA CUTLASS [32].

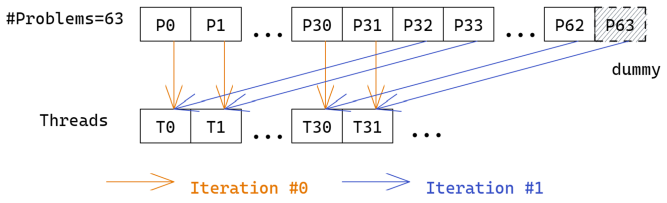



Fig. 7: Warp prefetching for grouped GEMM.

Grouped GEMM frequently checks with the built-in scheduler on the current task assignments, which leads to the runtime overhead. To address this issue, we propose an optimization over the built-in CUTLASS group GEMM scheduler. Figure 7 shows our optimization for the original CUTLASS grouped GEMM scheduler. Rather than asking one thread to compute the current tasks metadata, we have all 32 threads in a warp compute the tile indices to visit at one time. Therefore, we achieve 32X fewer scheduler visit overhead. In practice, this strategy brings a $\sim$10% improvement over the original CUTLASS grouped GEMM for standard BERT configurations. The prototype of this optimization has also been upstreamed to NVIDIA CUTLASS. We would refer an interested reader to [33] for detailed source codes.

In addition to optimizing the grouped GEMM scheduler, we fuse the memory footprints of softmax into two grouped GEMMs of MHA. Figure 8 shows the details of epilogue fusion for softmax reduction. A CTA computes an $M_C \times N_C$ sub-matrix. $M_C$ and $M_C$ are both set to 128 to maximize the performance of GEMM. Under the default CUTLASS threadmap assignment, there are 128 threads per CTA, and

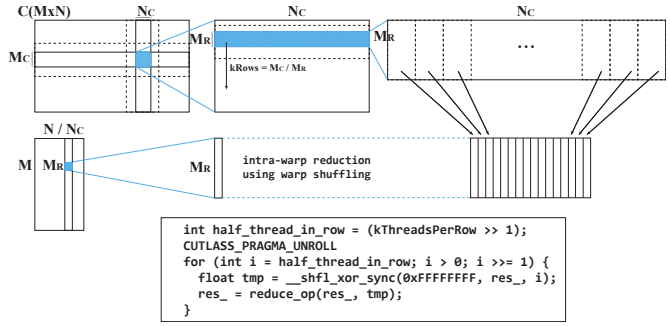



Fig. 8: Fused softmax reduction in grouped GEMM epilogue.

the threadmap is arranged as $8 \times 16$, where each thread holds a 128-bit register tile in each step. After the intra-thread reduction, the $M_R \times N_C$ ($8 \times 128$) sub-matrix is reduced to $8 \times 16$, with one reduced result held by one thread. We then conduct an intra-warp reduction to further reduce from the column dimension, which is implemented via CUDA warp shuffling for efficiency. Similar reductions (intra-thread followed by intra-warp reduction) are performed to compute both max and sum in epilogue. Once max and sum are both reduced, we store them to global memory.

The reduction in epilogue only provides us with partial reduction within a threadblock because cross-threadblock communication is impractical under the current CUDA programming model. Hence, we need to launch a separated lightweight kernel, as shown in Figure 6, to conduct the full reduction. In partial reduction, the target tensor of each attention unit is `seq_len` $\times$ `seq_len` while the full reduction just reduces a `seq_len` $\times$ `seq_len`$/128$. Therefore, the workload of full reduction is negligible to that of partial reduction In practice, the full reduction kernel only accounts for $\sim$ 2% of total execution time in fused MHA.

Once we have obtained the fully reduced $max$ and $sum$ vectors, we are ready to proceed element-wise transform $\frac{e^{x_{ij}} - max}{sum}$ on the first GEMM's output matrix. To hide the memory latency, we fuse these element-wise operations into the mainloop of the second GEMM. Algorithm III.2 presents our modifications (marked in red) of the original CUTLASS GEMM mainloop to enable softmax fusion. The original GEMM mainloop adopts the pipelining strategy to alleviate memory access latencies on both global memory and shared memory. For shared memory accesses, double register tiles are utilized to ensure that what is consumed in the current iteration has always been loaded in the previous iteration. For global memory accesses, a multi-stage loading strategy is employed with the help of the `cp.async` instruction of NVIDIA Ampere GPUs. The `cp.async` instruction allows loading data asynchronously from global memory to shared memory without consuming registers. Multiple such transactions can be proceeded concurrently, and a stage barrier ensures selected stages to be synchronized. The number of load stages (`kStages`) is a compile-time constant defined by a user. Similar to shared memory accesses, loading from global

**Algorithm III.2:** Mainloop fusion of grouped FMHA

```
Register Tiles:
WarpLoadedFragmentA warp_loaded_frag_A[2];
WarpLoadedFragmentB warp_loaded_frag_B[2];
WarpLoadedFragmentNormSum warp_loaded_frag_norm_sum;
Shared memory: (kStages + 1) shared-memory tiles for A and B
/* prologue */
Load k-invariant fused softmax tile to warp_loaded_frag_norm_sum
Prefetch kStages - 1 tiles of A to shared memory using cp.async
Prefetch kStages - 1 tiles of B to shared memory using cp.async
Prefetch a tile of A from shared memory to warp_loaded_frag_A[0]
Prefetch a tile of B from shared memory to warp_loaded_frag_B[0]
/* fused element-wise operation */
/* A = exp(A-max)/sum */
elementwise_transform(
 warp_loaded_frag_A[0],
 warp_loaded_frag_norm_sum);
/* mainloop */
for k to -kStages + 1 do
    /* Computes a warp-level GEMM */
    /* with pipelined load during iterations */
    for warp_mma_k = 0 to kWarpGemmIterations - 1 do
        Prefetch warp_loaded_frag_A[(warp_mma_k + 1) % 2]
        Prefetch warp_loaded_frag_B[(warp_mma_k + 1) % 2]
        /* fused element-wise transform */
        elementwise_transform(
         warp_loaded_frag_A[(warp_mma_k + 1) % 2],
         warp_loaded_frag_norm_sum);
        /* Computes a warp-level GEMM*/
        /* on data loaded in previous iteration */
        warp_mma(
         accum,
         warp_loaded_frag_A[warp_mma_k % 2],
         warp_loaded_frag_B[warp_mma_k % 2],
         accum);
        Prefetch a tile of A to shared memory using cp.async
        Prefetch a tile of B to shared memory using cp.async
```

memory is also pipelined to overlap memory latency with computation. Therefore, `kStages` pieces of shared memory buffers are needed under the multi-stage pipeline scheme. As shown in Algorithm III.2, we preload the k-invariant vectors $sum$ and $max$ in prologue, and conduct element-wise transform right after the matrix elements are loaded into registers. Since the fused vectors are loaded outside of the GEMM mainloop, only negligible overhead is brought into the baseline GEMM and the memory latency to perform element-wise transform is perfectly hidden with GEMM computations.

The baseline MHA is a computational chain containing a batched GEMM, a softmax, and another batched GEMM. The time and memory complexity of all these operations are quadratic in the sequence length. Because the padding-free algorithm directly reduces the effective sequence length, MHA with variable-length input also gains a direct improvement. Our fused MHA, which is explicitly designed to handle both short and long sequences, incorporates the padding-free algorithm to alleviate the memory overhead of the intermediate matrix in MHA caused by padding for variable-length inputs. Our highly optimized MHA outperforms the standard PyTorch MHA by 6.13X and further accelerates the single-layer BERT transformer by 19% compared to the previous step. As a result, this fully optimized version surpasses the baseline implementation in Figure 2 (a) by 60%. Since the remaining operations of a forward BERT transformer are all near-optimal GEMM operations, we conclude our optimizations at this step.

## IV. Evaluation

We evaluate our optimizations on an NVIDIA A100 GPU. The GPU device is connected to a node with four 32-core Intel Xeon Platinum 8336C CPUs, whose boost frequency is up to 4.00 GHz. The associated CPU main memory system has a capacity of 2TB at 3200 MHz. We compile programs using CUDA `11.6u2` with the optimization flag `O3`. We compare the performance of ByteTransformer with latest versions of state-of-the-art transformers, such as TensorFlow `2.8`, PyTorch `1.13`, Tencent TurboTransformer `0.5.1`, Microsoft DeepSpeed-Inference `0.7.7`, and NVIDIA FasterTransformer `5.1`. All the tensors benchmarked in this paper, unless specified, are in the half-precision floating-point format (FP16) to leverage tensor cores of NVIDIA GPUs. The variable sequence lengths in this section are generated randomly based on a uniform distribution with a range from 1 to the maximum length. We average the reported performance data over tens of runs to minimize fluctuations.

### A. Kernel fusion for layernorm and add-bias operations

As depicted in Figure 2, BERT transformer is composed of a series of GEMM and memory-bound operations. Since GEMM are accelerated by near-optimal vendor's libraries cuBLAS and CUTLASS, we focus on optimizing the functional modules that involve memory-bound operations.

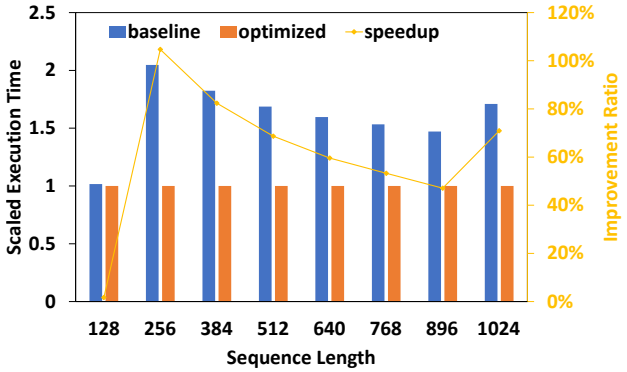



Fig. 9: Kernel fusion for add-bias and layernorm on a $(\texttt{batch\_size} \cdot \texttt{seq\_len}) \times \texttt{hidden\_dim}$ tensor. Here we profile for 16 batches with the hidden dimension fixed to 768 under the standard BERT configuration.

The result tensor needs to be added by the input tensor and normalized after projection and feed forward network of BERT transformer. Rather than launching two separated kernels, we fuse them into a single kernel and re-use data at the register level. In addition to kernel fusion, we leverage FP16 SIMD2 to increase the computational throughput of layernorm by assigning more workload to each thread. We normalize the execution time by that of the optimized layernorm and present the results in Figure 9: the improved version with kernel fusion provides us with a 69% improvement on average over the unfused baseline for sequence lengths ranging 128 to 1024.

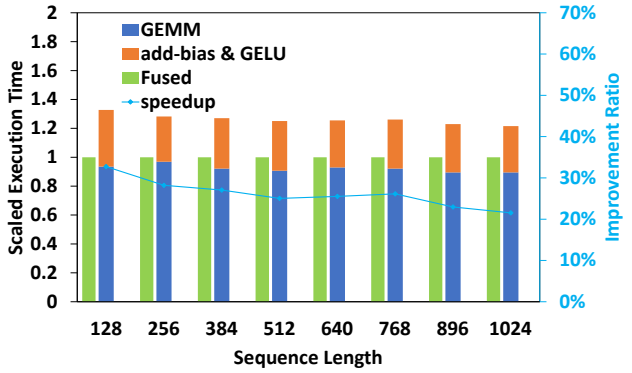



Fig. 10: Kernel fusion for GEMM, add-bias, and GELU. The shape of output tensor is $(\texttt{batch\_size} \cdot \texttt{seq\_len}) \times (\texttt{scale} \cdot \texttt{hidden\_dim})$. Here we profile for 16 batches with the hidden dimension and the scale factor fixed to 768 and 4 under the standard BERT configuration.

### B. Kernel fusion for GEMM and add-bias & activation

Regarding the GEMM, add-bias and activation pattern in BERT transformer, we also provide a fused kernel to reduce the global memory access. An unfused implementation is to call vendor's GEMM, store the output to global memory, and then load the result matrix from global memory for further element-wise operations. In our optimized version, when the result matrix of GEMM is held in registers, we conduct fused element-wise operations that re-use data at the register level. Once the element-wise transform (add-bias and GELU) is completed, we then store the results to the global memory. Figure 10 compares the performance of fused and unfused versions. In each clustered bar plot, the detailed execution time breakdown of the unfused implementation, normalized by the fused execution time (shown in the left bars), is shown in the stacked bar on the right. By fusing element-wise operations into the GEMM epilogue, we improve the performance by 24% on average for sequence lengths ranging 128 to 1024. It is worth mentioning that we feed *packed* tensors into both fused and non-fused kernels, such that the performance gain in Sec IV A and B are solely from kernel fusion.

### C. Optimizing multi-head attention

Figure 3 shows that MHA accounts for 22% - 49% of the total execution time. We optimize this key algorithm by fusing softmax into GEMMs without calculating for useless padded tokens under variable-length inputs. For short sequences, we hold the intermediate matrix in registers and shared memory. For long sequences, we adopt a grouped GEMM based fused MHA and fuse softmax operations into our customized GEMM epilogue and mainloop to hide the memory latency. In both implementations, the input matrices are accessed according to the position information obtained from the zero padding algorithm so that no redundant calculations are introduced.

Figure 11 compares the MHA performance for sequences shorter than 384. Here cuBLAS denotes the unfused implementation that calls cuBLAS for batched GEMM. The

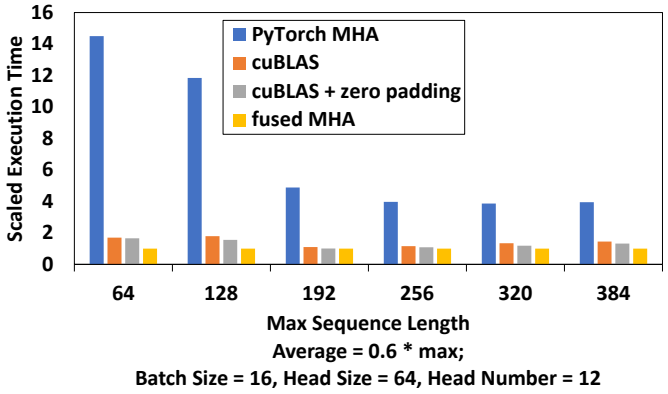



Fig. 11: Fused MHA for short sequences.

softmax operation between two batched GEMM can benefit from the zero padding algorithm, by only accessing unpadded tokens according to the known indices. This variant is denoted as *cuBLAS + zero padding* in the figure. cuBLAS batched GEMM improves the performance over stand PyTorch MHA by 5 folds while enabling the zero padding algorithm for softmax further improves the performance by 9%. Our MHA fully fuses the softmax and two batched GEMMs into one kernel, resulting in average speedups of 617%, 42%, and 30% over all three variants for variable sequence lengths ranging from 64 to 384.

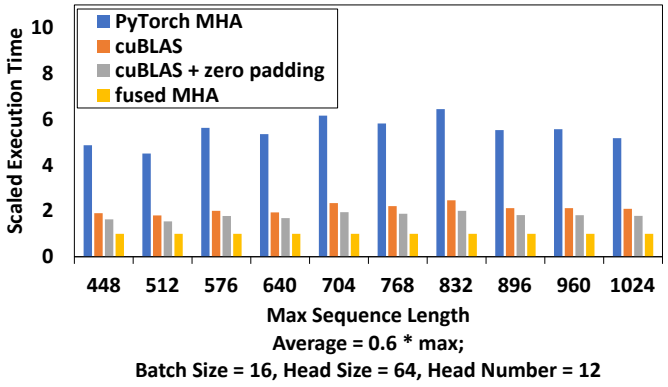



Fig. 12: Fused MHA for long sequences.

Figure 12 compares the performance of the MHA for sequences longer than 448. The cuBLAS batched GEMM triples the MHA performance over PyTorch, while eliminating wasted calculations in softmax further brings a 17% improvement. By introducing the high-performance grouped GEMM and fusing softmax into GEMMs, our fused MHA outperforms the variant MHA implementations by 451%, 110% and 79% for maximal sequence lengths ranging 448 to 1024, where the average sequence length is 60% of the maximum.

Figure 13 compares the scaled execution time of the FMHA module of our ByteTransformer against FlashAttention under the standard BERT setup. As shown in the figure, our FMHA presents advantages for small batch sizes (101% faster on average) while FlashAttention becomes more efficient for

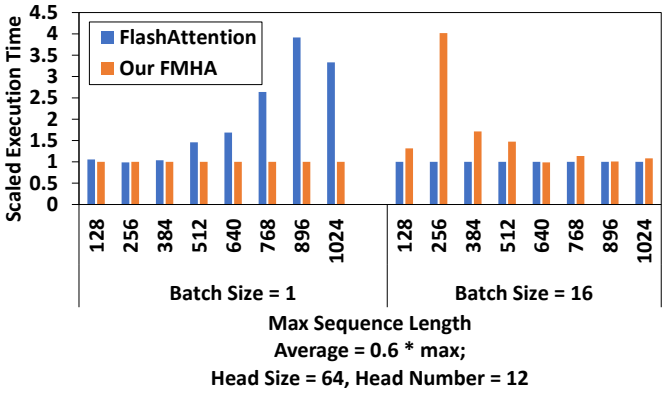



Fig. 13: Comparisons of our FMHA with FlashAttention.

large batch sizes (59% faster on average). This is because FlashAttention maps a whole attention unit to a threadblock, which, although allows for the complete preservation of the intermediate matrix of an attention unit within shared-memory for any sequence length, results in performance degradation when there are insufficient tasks assigned.

### *D. Benchmarking single-layer BERT transformer with step-wise optimizations*

Figure 14 compares the performance of a single-layer BERT transformer to reflect our step-wise optimizations. At each step, we add a new optimization upon the previous variant. The baseline transformer implements the workflow in Figure 2 (a) with padding. We then enable kernel fusion for adding bias and layernorm, which corresponds to *layernorm fusion* in the figure. The next step is to fuse adding bias and GELU into GEMM, denoted by *add bias & GELU fusion*. In order to avoid calculating padded tokens for the variable-length inputs, we further propose the zero padding algorithm as shown in Figure 2 (c). This is denoted by *rm padding* in the figure. Our optimized transformer includes our high-performance fused MHA, as well as all previous optimizations.

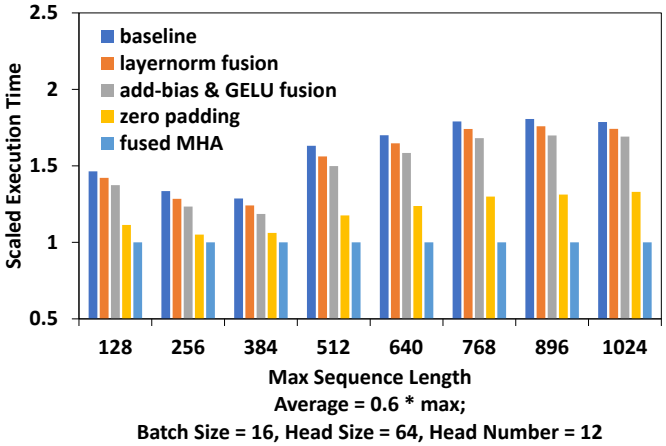



Fig. 14: Single-layer BERT transformer with step-wise optimizations. Each variant includes all previous optimizations.

Fusing adding bias and layernorm into one kernel improves the performance by 3.2%. Fusing adding bias and activation into GEMM epilogue further improves the performance by 3.8%. These two optimizations together improve the overall performance by 7.1%. After bringing in the zero padding algorithm, the redundant calculations are eliminated in most modules other than MHA. We observe a 24% improvement from the previous step. Finally, our fused MHA removes wasted calculations on padded tokens and enables an additional 20% improvement. To summarize, the final version achieves 60% improvement over the baseline version on single-layer BERT.

TABLE III. Single-layer BERT versus E.T. on A100.

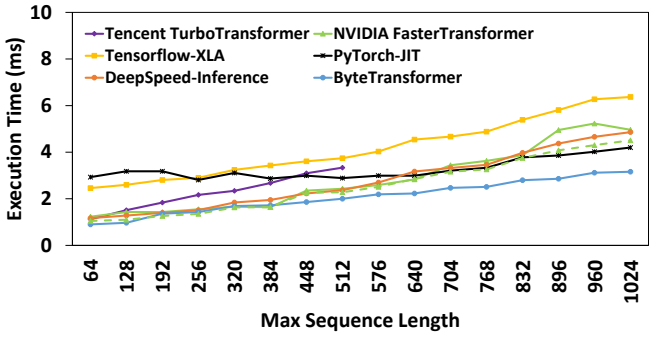
| Sequence Length | E.T. (ms) | ByteTransformer (ms) | Speedup |
|---|---|---|---|
| 256 | 0.25 | 0.07 | 3.57× |
| 1024 | 1.04 | 0.09 | 11.56× |

Table III compares the execution time for a single-layer, non-pruned BERT (batch size = 1) between E.T. and ByteTransformer, as E.T. has only open-sourced its single-layer, single-batch prototype. We achieve a speed-up of up to 11 times over E.T., which is optimized specifically for pruned models on legacy Volta GPUs. Since a pruned model can lead to significant reduction in total computations but with possible accuracy trade-offs, we do not include E.T. in our further end-to-end performance evaluations for non-pruned models on an A100 GPU for fairness and comparability.

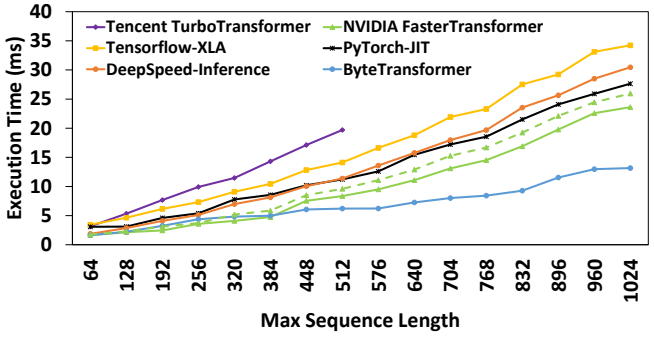
### *E. Benchmarking end-to-end performance of BERT*

The standard BERT transformer is a stacked structure of 12 layers of the encoder module. The output of each encoder module is utilized as an input tensor in the next iteration. Figure 15 shows the end-to-end performance of ByteTransformer and compares it against state-of-the-art transformer implementations: PyTorch with JIT, TensorFlow with XLA acceleration, Micorsoft DeepSpeed-Inference, NVIDIA FasterTransformer and Tencent TurboTransformer. We adopt the standard BERT transformer configuration for end-to-end benchmark: 12 heads, head size equal to 64 and 12 iterations (layers). We benchmark for cases whose batch sizes are equal to 1, 8 and 16 and change sequence lengths from 64 to 1024.

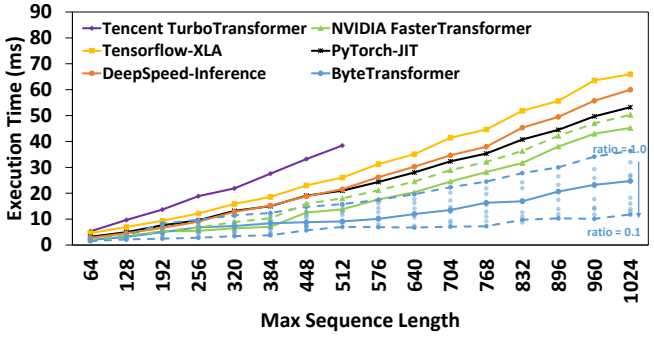
Compared with popular DL frameworks PyTorch, TensorFlow, and Microsoft DeepSpeed-Inference, our ByteTransformer achieves 87%, 131%, and 74% faster end-to-end performance on average. When benchmarking Tencent TurboTransformer, we turn on its `SmartBatch` mode to reach optimal batching performance. Since TurboTransformer only supports sequence lengths smaller than or equal to 512, we do not benchmark longer sequences for it. TurboTransformer regroups and pads similar sequences into a batch so it launches excessive kernels at the run-time. It is faced with significant performance degradation for models with large batch numbers and sequence lengths. NVIDIA FasterTransformer, although it supports long sequences regarding the functionality, its backend TensorRT fused MHA cannot be scaled to long sequences due to the limited register, its end-to-end efficiency cannot




(a) Batch size = 1




(b) Batch size = 8




(c) Batch size = 16

Fig. 15: End-to-end benchmark for standard BERT transformer, head size = 64, head number = 12, layer = 12, average sequence length = 0.6 * max sequence length.

be maintained when the sequence length becomes longer than 512. Experimental results in Figure 15 show that ByteTransformer outperforms TurboTransformer and FasterTransformer by 138% and 55% on average, respectively.

Figure 15 (c) further includes the end-to-end performance of ByteTransformer for average-to-maximum sequence length ratios ranging from 0.1 to 1.0. The upper dashed blue line represents the execution time of ByteTransformer at a ratio of 1.0, while the lower dashed line corresponds to a ratio of 0.1. Our padding-free algorithm reduces the runtime by up to 66% for a ratio of 0.1 compared to a fixed-sequence-length input. When disabling the support for variable-length inputs of FasterTransformer, as shown by the dashed green lines in Figure 15, we observe a moderate decrease in performance for larger batch sizes (batch sizes = 8 and 16) but an improvement in performance for a small batch size (batch size = 1). In contrast, our FMHA-enabled padding-free algorithm significantly improves the performance of the end-to-end BERT transformer for variable-length input with an average-to-maximum ratio of 0.6, outpacing NVIDIA FasterTransformer by a notable difference of 54% to 16%.

TABLE IV. Configurations of other BERT-like transformers.

| Model | layer number | head number | head size |
|---|---|---|---|
| ALBERT | 12 | 16 | 64 |
| DistilBERT | 6 | 12 | 64 |
| DeBERTa | 12 | 12 | 64 |

### *F. Extending to other BERT-like transformers*

We extend the optimizations on kernel fusion and the padding-free algorithm presented in our work to other BERT-like transformers, including ALBERT, DistilBERT, and DeBERTa. Table IV summarizes the model configurations, and readers can refer to [34]–[36] for more detailed information about their architectures. Figure 16 compares the performance of the ByteTransformer with state-of-the-art DL frameworks under these models. Following the setup for our demonstrated standard BERT benchmarks, the average sequence length is set to 60% of the maximal sequence length. TurboTransfomer only supports sequences shorter than 512, so its performance data for long sequences are not presented. FasterTransformer and TurboTransformer do not support DeBERTa, so their results are not included in that model. It is worth noting that TensorFlow encountered an out-of-memory error for sequence length 1024 in the DeBERTa model, resulting in this data point being excluded. For ALBERT and DistilBERT, our ByteTransformer on average outperforms PyTorch, TensorFlow, Tencent TurboTransformer, DeepSpeed-Inference, and NVIDIA FasterTransformer by 98%, 158%, 256%, 93%, and 53%, respectively. For the DeBERTa model, our ByteTransformer outperforms PyTorch, TensorFlow, and DeepSpeed by 44%, 243%, and 74%, respectively.

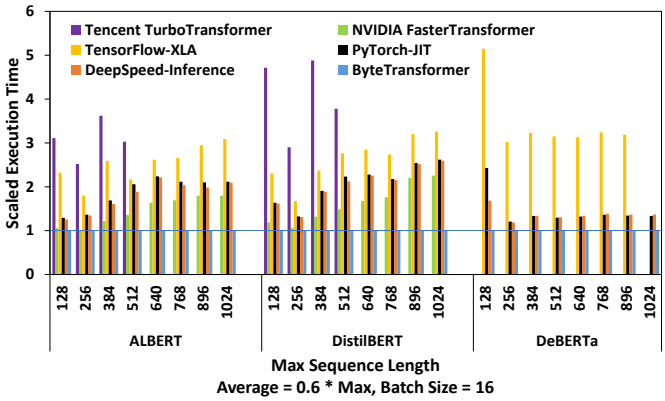



Fig. 16: End-to-end benchmark for other BERT-like models.

## V. Conclusions

We have presented ByteTransformer, a high-performance transformer optimized for variable-length sequences. ByteTransformer not only brings algorithmic level innovation that frees the transformer from padding overhead, but also incorporates architecture-aware optimizations to accelerate functioning modules of the transformer. Our optimized fused MHA, as well as other step-wise optimizations, together provide us with significant speedup over current state-of-the-art transformers. The end-to-end performance of the standard BERT transformer benchmarked on an NVIDIA A100 GPU demonstrates that our ByteTransformer surpasses PyTorch, TensorFlow, Tencent TurboTransformer, Microsoft DeepSpeed-Inference, and NVIDIA FasterTransformer by 87%, 131%, 138%, 74% and 55%, respectively. Moreover, we have shown that our optimizations are not specific to BERT, but can be applied to other BERT-like transformers, including ALBERT, DistilBERT, and DeBERTa. We are striving to make ByteTransformer completely open-source. This will allow the wider research community to benefit from our optimized implementation and to continue advancing the field. We are also dedicated to further expanding the presented strategies to accelerate a wider range of BERT-like transformer models, both in inference and training.

## References


[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," Advances in neural information processing systems, vol. 30, 2017.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," arXiv preprint arXiv:1810.04805, 2018.

[3] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," Advances in neural information processing systems, vol. 32, 2019.

[4] S. Edunov, M. Ott, M. Auli, and D. Grangier, "Understanding back-translation at scale," arXiv preprint arXiv:1808.09381, 2018.

[5] Q. Chen, H. Zhao, W. Li, P. Huang, and W. Ou, "Behavior sequence transformer for e-commerce recommendation in alibaba," in Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data, 2019, pp. 1–4.

[6] F. Sun, J. Liu, J. Wu, C. Pei, X. Lin, W. Ou, and P. Jiang, "Bert4rec: Sequential recommendation with bidirectional encoder representations from transformer," in Proceedings of the 28th ACM international conference on information and knowledge management, 2019, pp. 1441–1450.

[7] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever et al., "Language models are unsupervised multitask learners," OpenAI blog, vol. 1, no. 8, p. 9, 2019.

[8] J. Zeng, M. Li, Z. Wu, J. Liu, Y. Liu, D. Yu, and Y. Ma, "Boosting distributed training performance of the unpadded bert model," arXiv preprint arXiv:2208.08124, 2022.

[9] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell et al., "Language models are few-shot learners," Advances in neural information processing systems, vol. 33, pp. 1877–1901, 2020.

[10] 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, pp. 389–402.

[11] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2020, pp. 1–16.

[12] 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, pp. 3505–3506.

[13] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," arXiv preprint arXiv:1909.08053, 2019.

[14] X. Wang, Y. Xiong, X. Qian, Y. Wei, L. Li, and M. Wang, "Lightseq2: Accelerated training for transformer-based models on gpus," arXiv preprint arXiv:2110.05722, 2021.

[15] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., "{TensorFlow}: a system for {Large-Scale} machine learning," in 12th USENIX symposium on operating systems design and implementation (OSDI 16), 2016, pp. 265–283.

[16] Google, https://www.tensorflow.org/xla, Retrieved in 2022, online.

[17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., "Pytorch: An imperative style, high-performance deep learning library," Advances in neural information processing systems, vol. 32, 2019.

[18] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," arXiv preprint arXiv:1409.1259, 2014.

[19] NVIDIA , https://developer.nvidia.com/tensorrt, Retrieved in 2022, online.

[20] Y. Zhai, E. Giem, Q. Fan, K. Zhao, J. Liu, and Z. Chen, "Ft-blas: a high performance blas implementation with online fault tolerance," in Proceedings of the ACM International Conference on Supercomputing, 2021, pp. 127–138.

[21] K. Zhao, S. Di, S. Li, X. Liang, Y. Zhai, J. Chen, K. Ouyang, F. Cappello, and Z. Chen, "Algorithm-based fault tolerance for convolutional neural networks," IEEE Transactions on Parallel and Distributed Systems, 2020.

[22] Y. Zhai, M. Ibrahim, Y. Qiu, F. Boemer, Z. Chen, A. Titov, and A. Lyashevsky, "Accelerating encrypted computing on intel gpus," in 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2022, pp. 705–716.

[23] NVIDIA, https://github.com/NVIDIA/FasterTransformer, Retrieved in 2022, online.

[24] NVIDIA , https://developer.nvidia.com/cublas, Retrieved in 2022, online.

[25] S. Chen, S. Huang, S. Pandey, B. Li, G. R. Gao, L. Zheng, C. Ding, and H. Liu, "Et: re-thinking self-attention for transformer models on gpus," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021, pp. 1–18.

[26] R. Y. Aminabadi, S. Rajbhandari, M. Zhang, A. A. Awan, C. Li, D. Li, E. Zheng, J. Rasley, S. Smith, O. Ruwase et al., "Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale," arXiv preprint arXiv:2207.00032, 2022.

[27] PyTorch, https://pytorch.org/docs/stable/generated/torch.nn.MultiheadAttention.html, Retrieved in 2022, online.

[28] NVIDIA , https://github.com/NVIDIA/TensorRT/tree/main/plugin/bertQKVToContextPluginl, Retrieved in 2022, online.

[29] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," arXiv preprint arXiv:2205.14135, 2022.

[30] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," arXiv preprint arXiv:1606.08415, 2016.

[31] NVIDIA, https://github.com/NVIDIA/cutlass, Retrieved in 2022, online.

[32] NVIDIA , https://github.com/NVIDIA/cutlass/tree/master/examples/41_multi_head_attention, Retrieved in 2022, online.

[33] NVIDIA, https://github.com/NVIDIA/cutlass/blob/master/include/cutlass/gemm/kernel/grouped_problem_visitor.h#L203-L322, Retrieved in 2022, online.

[34] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," arXiv preprint arXiv:1909.11942, 2019.

[35] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," arXiv preprint arXiv:1910.01108, 2019.

[36] P. He, X. Liu, J. Gao, and W. Chen, "Deberta: Decoding-enhanced bert with disentangled attention," arXiv preprint arXiv:2006.03654, 2020.