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

Ayush Raina, 22148

April 9, 2025

Overview

This paper introduces **ByteTransformer**, a high performance transformer framework optimized for **variable** length sequences. Traditional transformer architectures pad sequences to a fixed length which leads to unnecessary computations and memory utilization. ByteTransformer applies a **padding free design** that improves the efficiency by eliminating the unnecessary operations on padded tokens. The authors demonstrate that how ByteTransformer outperforms existing state of the art transformer architectures implemented across several deep learning frameworks like **PyTorch** using **Fused Multihead Attention(MHA)** which is further optimized using **CUTLASS** kernels for NVIDIA GPUs.

Motivation

Transformer models have become the foundational models in NLP tasks such as machine translation, language modelling etc. However modern transformer architectures like **BERT, GPT-3** have billions of parameters resulting in massive computational overhead. Existing systems apply techniques like **JIT compilations** and **graph optimizations** but all these techniques still lack support for variable length input optimizations.

Methodology and Key Contributions

The authors have developed ByteTransformer, a high performance GPU accelerated transformer optimized for variable length inputs. It has been deployed for world class applications like **TikTok**. The main contributions are:

1. **Padding Free Design:** Inputs are packed into tensors without zero padding. They calculate **positioning offset vector** for all transformer operations to maintain correct attention behaviour and keeps the whole transformer pipeline free from calculations on zero tokens.

2. **Fused Multihead Attention Module:** They fuse MHA operations to reduce to quadratic memory and compute cost with respect to sequence length. This allowed to skip the unnecessary computations on padded tokens and improves the inference speed. This fused MHA is integrated with **NVIDIA CUTLASS** kernels.

3. **Superior Performance:** On NVIDIA A100 GPUs, fused MHA shows upto 6.13x speedup over standard PyTorch Attention. Regarding the end to end performance, ByteTransformer surpasses PyTorch, Tensorflow, Tencent Turbo-Transformer, Microsoft Deep Speed and NVIDIA FastTransformer by 87% , 131% , 138% , 74% and 55% respectively.

Designs and Optimizations

In this section we will discuss several key optimizations that ByteTransformer adopts over standard BERT encoder which involves generating the Q,K,V matrices through separate (GEMMs) which incurs significant kernel launch overhead. In ByteTransformer, these separate GEMMs are fused into a single batched GEMM operation. Additionally they pack the Q,K,V into contiguous memory to exploit locality.

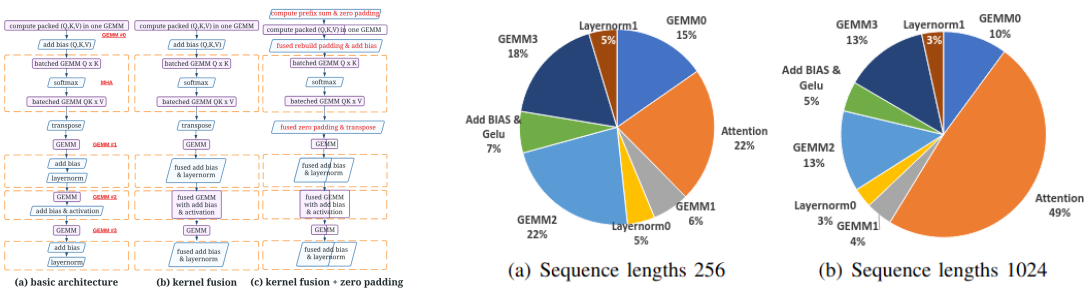

Figure 1: BERT Transformer Architecture, Optimizations and Performance for single layer

The authors profile a single layer BERT encoder layer (Figure 1.a) on NVIDIA A100 GPU to understand performance bottlenecks. Results show that compute bound operations like GEMMs dominate execution time at shorter sequences (61% for L = 256) while attention operations become more dominant at longer sequences (49% for L = 1024). Since GEMMs are already highly optimized in **cuBLAS**, the authors focus on optimizing the modules containing memory bound operations like **attention**(with softmax), **FFN**(with layernorm) and **add bias** followed by element wise activations. They optimize these operations by fusing distinct kernels and by reusing data in registers to reduce global memory accesses. Figure 1.b shows the pipeline with **add bias & activation** fused with GEMM kernel and fused **add bias & layernorm** kernel.

Fused Add Bias & Layernorm Kernel: After MHA, the resultant tensor needs to first be added to another tensor (bias) and perform layer normalization. The naive implementation involves 2 rounds of memory access to load and store the tensor. The Fused Kernel introduced accesses global memory only once to finish both **add bias** and **layernorm**. This fusion improved the **Single Layer BERT Transformer** performance by **3.2%** for sequence lengths ranging from 128 to 1024 in average.

Fused Add Bias & Activation Kernel: After GEMM operation, the resultant tensors undergoes addition of bias and **GELU** activation. This fused implementation instead of writing the GEMM output to global memory and then reloading it, applies bias and activation directly at **register level** using a customized **CUTLASS** epilogue. Using this we further improve the performance of **Single Layer BERT Transformer** by **3.8%**

Optimizing for variable length inputs (zero padding algorithm): Instead of padding shorter sequenes with zero tokens, this algorithm packs the input tensor containing variable length sequences into a contiguous block of memory and simultaneously computes the **positioning offset vector**. This offset vector acts as an index to the original sequences during subsequent transformer operations, allowing the model to effectively skip computations on what would have been zero-padded tokens. The process involves calculating the prefix sum of a mask matrix to identify valid tokens and then using this information to pack the input tensor. This padding-free algorithm further achieved a 24.7% acceleration of the BERT transformer when the average sequence length was 60% of the maximum length.

Optimizing Multi Head Attention: Even though the **zero padding algorithm** discussed above helps optimize many transformer operations like feed forward layers., it does not directly help in MHA module because batched GEMMs in MHA require uniform matrix shapes, so even if we pack our tensors in contiguous memory as discussed above, we still need to unpack and re introduce padding, bringing back inefficiency. To resolve this issue, authors propose a new **fused MHA Kernel** that works directly on contiguous packed tensors. The authors have proposed two algorithms one each for **shorter sequences** and **longer sequences**.

Unpadded Fused Multi Head Attention for short sequences: For short input sequences ($L \leq 384$), they hold the **intermediate matrix** (QK^T) in shared memory and registers throughout the MHA computation kernel to fully eliminate the **quadratic** memory overhead. They access the Q,K,V matrices according to positioning offset information calculated using prefix sums to avoid redundant computations on useless tokens.

Unpadded Fused Multi Head Attention for long sequences: For longer input sequences ($L > 384$), the key idea is to model each MHA computation as a separate sub problem in **Grouped GEMM.** Unlike traditional batched GEMM, grouped GEMM (from NVIDIA CUTLASS) allows arbitrary shapes for each problem, making it well-suited for handling variable-length sequences.

Fused Attention Computation: Each MHA operation involves computing: $P_i = Q_i K_i^T, softmax(P_i), O_i = S_i V_i$. GPU Execution workflow is as follows: **CTAs (Cooperative Thread Arrays, or threadblocks)** work on different subproblems. Each CTA computes a tile (chunk of output matrix). CTAs move through subproblems in waves, until all are done.

Optimization 1 - Warp Prefetching: Grouped GEMM frequently checks with the built-in scheduler on the current task assignments, which leads to the runtime overhead. In this case all 32 threads in a warp does the work together to achieve 32x fewer scheduler visit overhead.

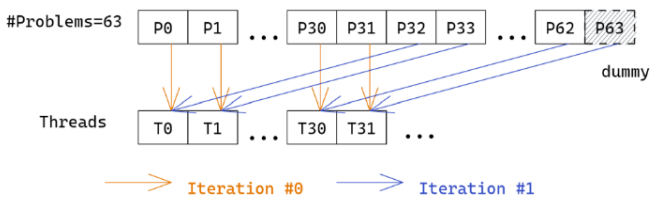

Figure 2: Warp Prefetching

Optimization 2 - Fusing Softmax Epilogue: After computing the first GEMM, a softmax operation is required. Instead of executing it separately, ByteTransformer fuses the softmax computation into the epilogue of first GEMM kernel. This fusion includes **intra thread reduction** followed by **intra warp reductions**. Partial results like **sum, max** are stored in global memory. Since full reduction across threadblocks is not possible within a single CTA, a lightweight

auxiliary kernel completes the global reduction. However, this kernel only contributes ∼2% of the total fused MHA runtime, making the epilogue fusion both efficient and practical.

Optimization 3 - Softmax-GEMM Fusion: To further reduce the memory traffic and hide latency, ByteTransformer fuses the element wise softmax transformation into the main loop of second GEMM, reusing the pipeline infrastructure already present in CUTLASS

Results

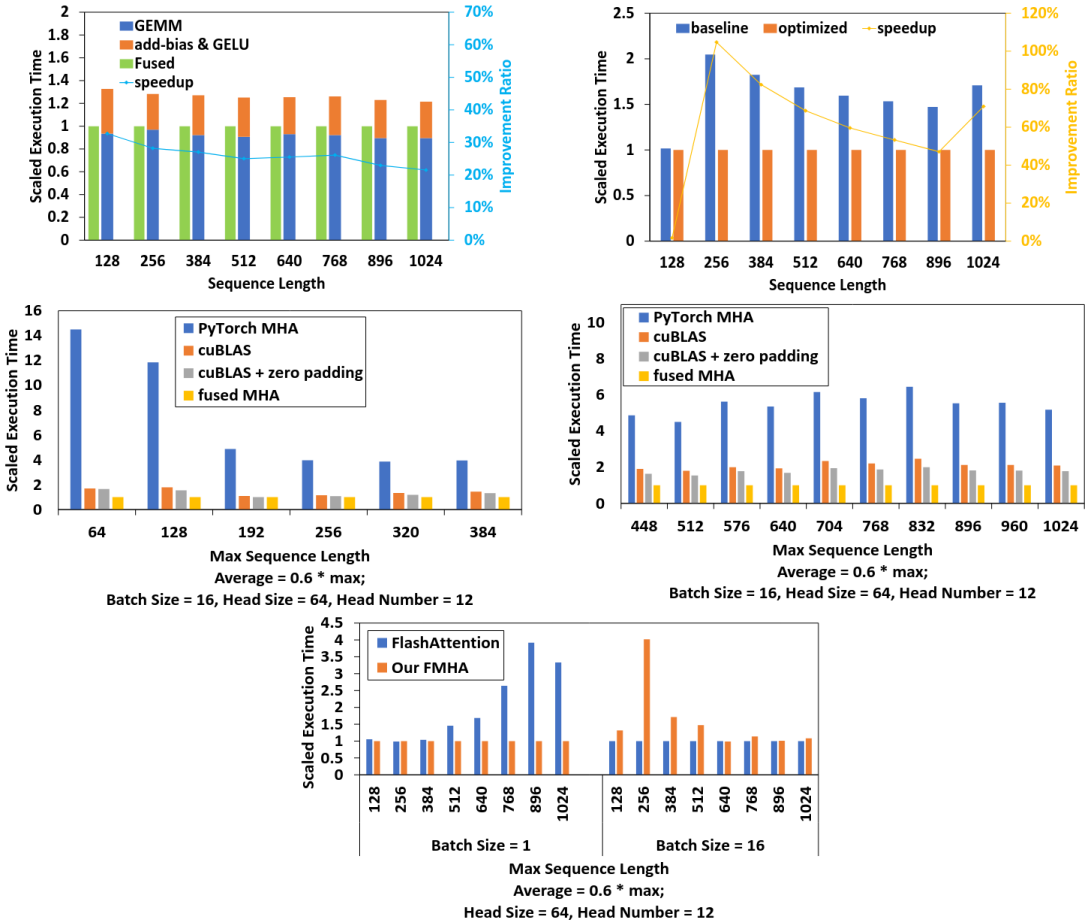

Figure 1 shows results for **Fused GEMM with Add Bias & Activation**, Figure 2 shows results for **Fused GEMM with Add Bias & Layernorm**, Figure 3 shows results for **Fused MHA for Short Sequences**, Figure 4 shows results for **Fused MHA for Long Sequences** and Figure 5 is for comparison with **Flash Attention**

Critical Analsis

A key strength of ByteTransformer is its hardware-conscious kernel design, which effectively utilizes shared memory, cp.async pipelining, and warp-level parallelism. The proposed padding-free attention and fused softmax-GEMM kernels eliminate unnecessary memory movement and padding overhead, achieving significant real-world speedups. The warp-synchronous grouped GEMM optimization is particularly impactful, reducing scheduling overhead and boosting throughput.

However, the paper primarily evaluates kernel-level performance and does not present end-to-end model benchmarks (e.g., full finetuning or downstream accuracy). This limits visibility into the overall system impact. Also, the optimizations focus on encoder-style architectures, with limited discussion on applicability to decoders or autoregressive models.

As a potential extension, future work could focus on training-time fusion with dropout and residuals or adapting the optimizations for quantized models. This would make the approach more broadly useful for both deployment and efficient training across diverse transformer variants.