EASY AND EFFICIENT TRANSFORMER: SCALABLE INFEERENCE SOLUTION FOR LARGE NLP MODEL

Gongzheng Li*1 Yadong Xi*+1 Jingzhen Ding Duan Wang Bai Liu Changjie Fan Xiaoxi Mao Zeng Zhao

ABSTRACT
Recently, large-scale transformer-based models have been proven to be effective over various tasks across many domains. Nevertheless, putting them into production is very expensive, requiring comprehensive optimization techniques to reduce inference costs. This paper introduces a series of transformer inference optimization techniques at the algorithm and implementation levels. These techniques include a pre-padding decoding mechanism that improves token parallelism for generation and highly optimized kernels designed for very long inputs and large hidden sizes. On this basis, we propose a transformer inference acceleration library – Easy and Efficient Transformer (EET), which has a significant performance improvement over existing libraries. Compared to Faster Transformer v4.0’s implementation for transformer decoder layer on A100, EET achieves an average 2-4.5x state-of-art speedup. EET is available at https://github.com/NetEase-FuXi/EET. A demo video is available at https://youtu.be/22UPcNGcErg.

1 INTRODUCTION
In recent years, transformer-based models have achieved impressive results in multiple domains including natural language processing (Vaswani et al., 2017; Devlin et al., 2019; Raffel et al., 2020; Brown et al., 2020), computer vision (Jiang et al., 2021; Dosovitskiy et al., 2020) and speech processing (Baevski et al., 2020; 2021) etc. Kaplan et al. proposed the scaling law, which argues that the loss of a neural language model scales as a power-law with model sizes, dataset sizes, and the amount of compute used for training (Kaplan et al., 2020). Many following works have empirically supported it (Brown et al., 2020; Zhai et al., 2021).

However, the mega-sized models are notoriously costly for deployment. For example, GPT-2 medium model (Radford et al., 2019) (hidden_units=1280, layer_num=36, precision=fp16) takes about 10s to make an inference with prompt length of 512 and sequence length of 1024 on an RTX 2080ti. This obvious drawback has spurred multiple approaches proposed to improve the inference efficiency, such as knowledge distillation (Hinton et al., 2015; Jiao et al., 2020), model pruning (Voita et al., 2019), and quantization (Shen et al., 2019). Apart from these efforts, a lot of attention has also been paid to optimizing the transformer layer’s CUDA implementation for better hardware utilization. Previous works include TensorRT (NVIDIA, 2021b), Faster Transformer(FT) (NVIDIA, 2021a) and Lightseq (Wang et al., 2021). They have implemented multiple optimization techniques, including kernels fusion, gemm optimization, quantization, etc. However, they still come up with several limitations. For example, they have not fully exploited the parallelism of the input sequences, which leads to inefficient utilization of GPU computing power; some could not support the model hidden size and input sequence length above 1024; besides, applying them in applications is also non-trivial.

In this paper, we propose a novel transformer inference acceleration library designed basing on PyTorch(Paszke et al., 2019), the Easy and Efficient Transformer(EET). In EET, we implement a bunch of optimization techniques. Firstly, we propose a novel decoding mechanism called pre-padding decoding to reduce the time complexity of decoding for the auto-regression model. Secondly, we implement custom CUDA kernels to avoid explicit matrix multiplications of attention masks and attention weights. Finally, we extend all kernels to support a larger model size up to 12288 and longer sequence up to 4096 and achieve 2-4.5x speedup compared to the Faster Transformer and 7-10x speedup compared to Fairseq on average.

While deploying a PyTorch model with TensorRT or Faster Transformer is sophisticated, the deployment workflow with EET is greatly simplified. EET provides both op-level and model-level APIs. We also integrated a python based web server to improve the service capability with some essential features, including dynamic batching, variable input length, reordered incremental states, and early stop, which further improve the service efficiency.
The rest of the paper is organized as follows. Beginning with the proposed decoding mechanism, we introduce its algorithm advantages over the existing schemes; next, we illustrate our CUDA optimization made for scalability and efficiency, and our memory management strategies made for practicality; furthermore, a friendly pipeline to production is described for usability; at last, we demonstrate the three features (scalability, efficiency, practicality) from the experiments for different models on different NVIDIA GPUs.

2 Pre-padding Decoding

Teacher forcing is essential for fast convergence at training of auto-regressive models. It uses the the ground truth as the input for each time step, rather than the output of the network. Thus the model processes an entire batch in one pass. However, at inference, the decoder will have to feed its previous predictions back to itself to make the next prediction. This discrepancy between training and inference may lead to poor decoding performance because every token’s intermediate states will be recalculated for all its successive predictions. To address this issue, incremental decoding proposed in Fairseq (Ott et al., 2019) saves the intermediate states calculated in every step and reuse them for successive predictions. The size of intermediate states will increase as the tokens are generated, so this decoding mechanism is called incremental decoding.

Though incremental decoding is helpful for improving decoding efficiency, processing the tokens in the input sequentially instead of parallelly is inefficient. As the tokens in the input sequence are independent to each other, they can be fed into the model as a whole and get processed in one pass, similar to the teacher forcing. This style makes full use of GPU parallel processing power. Then intermediate states of the input tokens can still be cached for later incremental decoding. In this way, the number of decoding steps is:

\[
\text{steps} = 1 + s - p
\]

Where the \( s \) is the output length (sequence length), and the \( p \) is the input length (prompt length). Compared to incremental decoding, it skips the decoding steps of \( p \) times.

If the inputs in a batch are of different lengths, and extra padding process needs to be applied. The most common practice is to append padding tokens like [pad] to the end of input sequences until they reach the length of the longest sequence in the batch. Instead of this post-padding style, we argue that the pre-padding style, which appends padding tokens to the front of input sequences, has several benefits:

a) With post-padding, the padding tokens will separate the generated tokens and input tokens, leading to inaccurate relative positions. Pre-padding overcome this trouble without amendment to position embedding module; b) only affects the tokens to the right of padding tokens while post-padding makes it impacts on both sides, thus simplifies the underlying C++/CUDA implementation; c) pre-padding is more memory-efficient. We can use a single value instead of many explicit padding tokens to indicate the padding position in a sequence. As shown in Figure 1, the model processes the pre-padded batch in one pass and generates tokens with incremental decoding.

While the time complexity of the vanilla masked multi-head attention is: \( O(n^2 \times d) \) and the time complexity of the incremental decoding is: \( O(n \times d) \), the time complexity of our pre-padding decoding is: \( O((n - p) \times d) \). Here the \( n \) is the output length, the \( d \) is the model size, and the \( p \) is the input length. The pre-padding fashion gives an advantage of decoding efficiency over commonly used post-padding.

3 High Performance Kernels

Faster Transformer implements highly optimized CUDA kernels for transformer inference. With reference to its implementation, We design and implement our custom kernels, making further optimization with the considerations below:

1. Because padding tokens do not affect the final results,
avoiding the participation of padded tokens in multi-head attention instead of simply applying padding masks can significantly reduce the computational overhead.

2. Although sequence mask is necessary for the multi-head attention, reconstructing a mask that varies with the input length is time-consuming.

3. The hidden sizes and input lengths of latest large scale pre-trained models easily exceed 1024. It is necessary to extend these kernels to support large hidden sizes and input lengths elegantly and efficiently.

To remove previous mentioned masks, we redesign the kernels, and name the mechanism mask fusion. To extend all the kernels to support the model size or sequence length greater than 1024, we improve the CUDA thread structure, and name the method as thread block folding. In the following sections we describe these two methods in detail.

3.1 Mask Fusion

Both the sequence masks and padding masks characterize the positions of the tokens in a sequence. Meanwhile, each CUDA thread also has a unique positional index. So we can map every token in the masked multi-head attention to a respective thread or block in the CUDA kernels. Sequence mask is avoided by comparing whether the CUDA position of the query token being processed is larger than the CUDA position of the key token; padding mask is avoided by starting the valid calculations from the padding offset when sequentially processing each token. Therefore, we transform the mask computation to logical operation by CUDA thread index comparison operation. Thus there is no need to store any explicit functional parameters of the masks and the computation overhead of masking operation is avoided. We also show pseudo-code in Algorithm 1, and the C++ implementation could be found in our open-sourced repository (EET).

3.2 Thread Block Folding

Latest large scale models often have model sizes and input lengths larger than 1024. For example, the standard GPT-3 has a model size of 12288 and input length of 2048. However, because the CUDA block only supports a maximum thread number of 1024, most inference frameworks, such as FT v3.1 and Lightseq, implemented kernels that restrict the model size and input length up to 1024, leading to limited availability.

To deal with very large model sizes and sequence lengths, we propose to use several blocks to simulate a large block, shown as Figure 2. Imagine a virtual block large enough to hold all the tasks, then we can fold it once to create two

---

**Algorithm 1** transformer self attention with mask fusion

**Input:** qk, paddingLen, seqLen, batch, headNum

**Output:** the attention weights back to qk

CUDA Initialize: grid ← (batch * headNum)

CUDA Initialize block ← (seqLen)

batchId ← blockIdx.x/headNum

padLen ← paddingLen[batchId]

qkOffset ← blockIdx.x * seqLen + seqLen

qkOffset ← qkOffset + padLen * seqLen

s ← padLen \{start at first non-pad\}

e ← seqLen \{end at last token\}

reduceMax ← −inf

reduceSum ← 0

for i = s to e do

    position ← qkOffset + threadIdx.x

    data ← qk[position]

    u ← padLen \{upper boundary\}

    l ← i \{lower boundary\}

    if l < threadIdx.x < u then

        reduceMax ← blockReduceMax(data)

        reduceSum ← blockReduceSum(data)

        data ← softmax(reduceMax, reduceSum)

    end if

    qk[position] ← data

end for

blocks, with each block having half the size of the original block. We can repeat the process until the sub-blocks size satisfies the CUDA constraint. Thus, the large model sizes or input lengths can be handled correctly, and a new CUDA thread dimension is created to manage the folding procedure. We call this method Thread-Block-Folding, which allows us to extend any kernel to any model size and any sequence length with minimum changes and non-degraded performance. For instance, assuming the model size is 1280, we fold it once and create two half-size blocks, then the data can be assigned into two separate blocks with 640 threads in each.

We introduce a folding coefficient to characterize the number of folding. Given the model size h, the folding coefficient t and the number of threads n in one block is defined as:

\[
t = 2^\left\lceil \frac{\log h}{\log 2} \right\rceil - 1; \quad n = \frac{h}{2^t}
\]

As for simplicity, thread block folding only slightly impacts the basic CUDA thread grid structure, which adds a new dimension for the block. As for efficiency, the minimum thread number is 512 when the model size or input length is larger than 1024 and makes full use of thread parallelism. The sequence expansion process is similar to the model expansion process. Finally, we support the model size no greater than 16384 and sequence length no greater than 4096.
4 Dynamic Memory Manager

Inference is much more sensitive to latency compared to training. Model parallelism (Shoeybi et al., 2020), and pipeline parallelism (Huang et al., 2019) are not optimal solutions for inference, due to the communication overhead introduced by tensor slicing or layer split, which is significant even with the support of NVLink and GPUDirect. To reduce the latency and hardware requirements for online service, we should try to load the complete model on a single card. To load very large models, minimizing the memory footprint is necessary. So we propose a dynamic memory management strategy for this issue.

Except for the model weights, the memory footprint includes the buffers and the caches. It is hard to lower the memory footprint of weights because they are inherent to the model. Similarly, the $K/V$ caches are also hard to compress because they are pre-allocated to avoid runtime memory requests, depending on the model size, maximum batch size, and maximum sequence length. Whereas the activation cache and the buffers used to store the operator’s results are compressible. Hence our dynamic memory management strategy mainly focuses on the activation caches and the buffers. Considering the size of the weights:

$$S_w = \frac{v}{l} + 13 + 12 \times h \times h \times l$$

where $v$ is the vocab size, $l$ is the layer number, and $S_w$ is the size of weights.

4.1 Cache Reuse

The cache contains $K/V$ caches and activation caches. In pre-padding decoding, the keys and the values for every step are stored for next step’s attention calculation. The maximum size of $K/V$ caches is predictable because we can determine the maximum batch size and decoding steps at the start of the running instance. We allocate the maximum needed memory in advance to reduce the forward latency, avoiding malloc overhead and memory corruption.

Different from $K/V$ cache, the activation is useless after we have calculated and passed it to the next layer. The memory for these activation can be reused across different layers and different operators. As shown in Figure 3, EET could reuse the activation caches in the following cases:

1. The embedding operator shares the cache with the feedforward operator and the final output. Yet the attention operator holds another cache because of the residual connection.

2. The cache for input sequences can be reused by the decoded tokens. The maximum size is determined by the maximum input length.

3. The cache can be reused across different layers.

We use the following notations: $b$, the maximum batch size; $s$, the maximum sequence length; $p$, the maximum prompt length; $h$, the hidden units; $l$, the layer number. The total activation cache size is:

$$2 \times b \times h \times l$$

The total $K/V$ cache size is:

$$2 \times b \times h \times s \times l$$
4.2 Buffer Reuse

The continuous CUDA kernels are not always fused, especially when it comes to Cublas GEMM calls. So we need the buffers to store the returns for those non-fused kernels. It is complicated and inefficient to manage the buffers manually. We developed a dynamic buffer manager to avoid the tedium of manual design and dynamically find an optimal memory allocation.

We maintain a list of buffers and use different strategies within and across modules to improve memory utilization. When within modules, we reuse the buffer only when the request size is identical to an idle buffer in the list, preventing memory fragmentation. When across modules, we reuse the buffer when the request size is smaller than any idle buffer in the list, avoiding duplicated malloc. The decision process is shown in Figure 5.

The total buffer size is:

\[ b \times p \times (6 \times h + n \times p) \]

where \( b \) is the batch size, \( p \) is the input length, \( h \) is the hidden size and \( n \) is the head number.

5 EET In Production

EET provides two levels of APIs: op-level and model-level, shown in Table 1 for more flexible model architecture. In addition, EET can be easily integrated into existing Fairseq and Transformers (Wolf et al., 2020) projects by simply replacing several specified files. End-users will enjoy the speedup without any code changes and any model conversion.

Dynamic batching is a critical way to increase the throughput of a deployed online service by merging multiple client requests into one batch and making inferences in parallel. EET implements three key features to make it available:

**Variable batch size.** EET sets a maximum batch size and allocates all the GPU memory in advance, allowing every batch size smaller than the maximum batch size at runtime.

**Variable input length.** To merge different requests into a single batch, EET sets a maximum sequence length and allocates all the GPU memory in advance likewise, then padding the input sequences to the front.

**Early stop.** At generation, some of the input sequences would be finished early. EET supports reordering incremental states and avoids some of the sequences in the batch to continue occupying computational resources after they finish.

EET integrates an open-sourced python middleware, Service-Streamer (ShannonAI, 2021), to provide web service with dynamic batching. Deploying a web API service with EET is much more convenient compared to TensorRT and Triton\(^1\), which require a complex workflow. Our deploying workflow is demonstrated in Figure 6.

\(^1\)https://developer.nvidia.com/nvidia-triton-inference-server
Table 1. Model-level and Op-level APIs

| APIs Types       | API Names                                      |
|------------------|------------------------------------------------|
| MODEL APIs       | GPT-2; BERT                                    |
| Op APIs          | EMBEDDING; MULTI-HEAD-ATTENTION; MASKED-MULTI-HEAD-ATTENTION; CROSS-MULTI-HEAD-ATTENTION; FFN; LAYERNORM |

Table 2. Configurations A and B

| Configuration A | Configuration B |
|-----------------|-----------------|
| BATCH SIZE      | 4               |
| MODEL SIZE      | 1024            |
| MAX PROMPT      | 1024            |
| MAX SEQUENCE    | 1024            |
| DATATYPE        | FP16            |

6 Experiments

6.1 Memory distribution

Firstly, we try to analyze the memory distribution theoretically. Given the batch size 16, the maximum sequence length 1024, the vocab size 13672, we plot the memory distribution of the hidden size of 1024 and 4096 with layer numbers 24 and 40, respectively, in Figure 7. Regardless of the hidden size, we can find that model weights and K/V cache occupy most memory. The activation cache and the buffers only take up a small part, which shows the effectiveness of our dynamic memory management strategy.

6.2 Speedup for GPT-2 layer with different lengths

To prove the efficiency, we tested the performance of EET over GPT-2 on NVIDIA A100, 2080ti, and 3090, comparing it with Fairseq, LightSeq, and Fast Transformer. The configuration is as Table 2 configuration A.

Both Fairseq and Faster Transformer(v3.1) only implement
The recent version of Faster Transformer (v4.0) also introduces the parallel decoding of the input sequences and incremental decoding for generation. However due to our optimal kernels, EET still have some performance advantages. Figure 11 shows that EET achieves about 1.4-2.6x speedup compared to Faster Transformer (v4.0).

Except for the kernel optimization, pre-padding mechanism also gives EET some advantages. Pre-padding decoding could bring further performance boost when lengths of the input sequences in the batch are uneven. Because the Faster Transformer (v4.0) uses the minimum length of the prompts for full decoding, but the EET uses the maximum length of them. For example, if there is a batch containing sequences of different lengths like \((5, 1, 4, 10)\), the final prompt length used for parallelism is 1 in the Faster Transformer. In contrast, it is 10 in our EET. Figure 12 shows that we make 2.8-4.4x speedup with the prompt fixed to 512 and other configurations keeping the same as the Table 2 configuration B.

Unlike Fairseq and Faster Transformer v4.0, LightSeq only supports model sizes that are smaller than 1024, we also make a comparison here as a supplement. Figure 13 shows that we make 0.8-2.4x speedup when we set the model size to 768 and 1024.

### 6.3 Speedup for transformer decoder layer with different model sizes

| Configuration C | Configuration C |
|-----------------|-----------------|
| Batch size      | 4 / 8           |
| Prompt          | 512             |
| Max sequence    | 1024            |
| Datatype        | FP16            |

To prove the scalability, we test the performance with different model sizes, and the basic configuration is as configurations C shown in Table 3. Figure 14 and Figure 15 show the speedup compared to Fairseq and Faster Transformer (v4.0) respectively. Compared to Fairseq, EET achieves about 2.25-7.5x speedup; Compared to Faster Transformer (v4.0), EET achieves about 1.7-4.6x speedup.

The acceleration ratio decreases as the model size increases due to the increased ratio of matrix multiplication in the inference. Nevertheless, benefit with thread block folding, EET can still deliver significant speedup with very large...
7 Conclusion

This paper describes a comprehensive bunch of optimization techniques for transformer inference acceleration exploiting both algorithmic and GPU hardware features. These techniques are packed in the EET, a library dedicated to inference acceleration for large transformer-based models and long input lengths. EET has a 1.5-4.5x speedup for the GPT-2 layer and a 1.0-1.27x speedup for the Bert layer compared to Faster Transformer. We have open-sourced EET on Github. We will continue to improve and keep it up-to-date.

References

Baevski, A., Zhou, H., Mohamed, A., and Auli, M. wav2vec 2.0: A framework for self-supervised learning of speech representations. arXiv preprint arXiv:2006.11477, 2020.

Baevski, A., Hsu, W.-N., Conneau, A., and Auli, M. Unsupervised speech recognition. arXiv preprint arXiv:2105.11084, 2021.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners, 2020.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An image is worth 16x16 words: Transformers for image recognition at scale, 2020.

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network, 2015.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

Jiang, Y., Chang, S., and Wang, Z. Transgan: Two transformers can make one strong gan, 2021.

Jiao, X., Yin, Y., Shang, L., Jiang, X., Chen, X., Li, L., Wang, F., and Liu, Q. Tinybert: Distilling bert for natural language understanding, 2020.
Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models, 2020.

NVIDIA. "faster transformer", 2021a. URL https://github.com/NVIDIA/DeepLearningExamples/tree/master/FasterTransformer.

NVIDIA. "tensorrt", 2021b. URL https://github.com/NVIDIA/TensorRT.

Ott, M., Edunov, S., Baevski, A., Fan, A., Gross, S., Ng, N., Grangier, D., and Auli, M. Fairseq: A fast, extensible toolkit for sequence modeling. In Proceedings of NAACL-HLT 2019: Demonstrations, 2019.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d’Alch´e-Buc, F., Fox, E., and Garnett, R. (eds.), Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research, 21 (140):1–67, 2020. URL http://jmlr.org/papers/v21/20-074.html.

ShannonAI. Shannonai-service-streamer, 2021. URL https://github.com/ShannonAI/service-streamer.

Shen, S., Dong, Z., Ye, J., Ma, L., Yao, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. Q-bert: Hessian based ultra low precision quantization of bert, 2019.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2017.

Voita, E., Talbot, D., Moiseev, F., Sennrich, R., and Titov, I. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned, 2019.

Wang, X., Xiong, Y., Wei, Y., Wang, M., and Li, L. Lightseq: A high performance inference library for transformers, 2021.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Transformers: State-of-the-art natural language processing. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp. 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

Zhai, X., Kolesnikov, A., Houlsby, N., and Beyer, L. Scaling vision transformers. arXiv preprint arXiv:2106.04560, 2021.