Easy and Efficient Transformer: Scalable Inference Solution For Large NLP Model

Gongzheng Li[1]*, **Yadong Xi**[1]*, **Jingzhen Ding**[1], **Duan Wang**[1], **Ziyang Luo**[2], **Rongsheng Zhang**[1], **Bai Liu**[1], **Changjie Fan**[1], **Xiaoxi Mao**[1]†, **Zeng Zhao**[1]†

[1] Fuxi AI Lab, NetEase Inc., Hangzhou, China

[2] Department of Computer Science, Hong Kong Baptist University, Hong Kong SAR, China

{ligongzheng,xiyadong,maoxiaoxi,hzzhaozeng}@corp.netease.com

Abstract

Recently, large-scale transformer-based models have been proven to be effective over various tasks across many domains. Nevertheless, applying them in industrial production requires tedious and heavy works to reduce inference costs. To fill such a gap, we introduce a scalable inference solution: **Easy and Efficient Transformer (EET)**, including a series of transformer inference optimization at the algorithm and implementation levels. First, we design highly optimized kernels for long inputs and large hidden sizes. Second, we propose a flexible CUDA memory manager to reduce the memory footprint when deploying a large model. Compared with the state-of-the-art transformer inference library (Faster Transformer v4.0), EET can achieve an average of 1.40-4.20x speedup on the transformer decoder layer with an A100 GPU.

1 Introduction

In recent years, transformer-based models have achieved impressive performance across variant domains, such as 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). The scaling law proposed by Kaplan et al. (2020) indicates that the validation PPL of a neural language model scales as a power-law with model sizes, dataset sizes, and the amount of training computation. Such law is corroborated empirically by many following works (Brown et al., 2020; Zhai et al., 2021).

However, the mega-sized models are notoriously expensive for deployment in the industry. For example, GPT-2 medium model (700M parameters (Radford et al., 2019)) spends up to 10s to generate 512 tokens given a prompt with the length of 512 on an RTX 2080ti GPU, which is not allowed in the industrial application. Multiple approaches have been proposed to solve such problems, including 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 works, much attention has also been paid to optimizing CUDA implementation of a transformer layer for better hardware utilization. Previous works (e.g.: TensorRT (NVIDIA, 2021b), LightSeq (Wang et al., 2021) and Faster Transformer (FT) (NVIDIA, 2021a)) have implemented many optimization techniques, including kernels fusion, gemm optimization, quantization, etc. However, these works still have several limitations. TensorRT only contains the multi-head attention(MHA) operation, lacking a complete transformer model. LightSeq cannot support the model hidden size and input sequence length over 1024. FT contains some performance flaws which need to be improved.

In this paper, we propose a novel transformer inference acceleration library, **Easy and Efficient Transformer (EET)** . First, we implement custom CUDA kernels to avoid explicit matrix addition of attention and padding masks with attention weights. As a result, the attention mask matrix is no longer required, while FT spends overhead to initialize an attention mask on the CPU and push it to CUDA. In addition, compared with FT, padding masks are no longer needed in computation, leading to additional performance improvement. Second, we propose a new method, *thread block folding*, to extend all kernels to support a larger model size up to 12288 and a longer sequence up to 4096. For FT, it directly assigns the thread number in a CUDA block, which may hurt the parallel efficiency. Finally, we design a dynamic CUDA memory management mechanism to reduce the CUDA memory occupation for the same model size, while FT needs to manually allocate memory usage.

We have conducted comprehensive experiments

* Equal contribution

† Corresponding Author

Proceedings of NAACL-HLT 2022: Industry Track Papers, pages 62 - 68

to compare EET with Fairseq,[1] LightSeq and FT. In our experiments, EET achieves about 4.48-20.27x and 4.30-27.43x speedup over Fairseq on 2080ti and A100 respectively. When we set the model size to 768 and 1024 on 2080Ti, EET makes 0.82-2.46x speedup over LightSeq. Compared to FT(v3.1), EET achieves about 1.21-6.30x and 1.62-8.16x speedup on 2080ti and A100 respectively. Compared to FT(v4.0), EET achieves about 1.40-4.20x speedup on A100. The remarkable experimental results corroborate the effectiveness of our EET.

2 Custom Kernels

FT (NVIDIA, 2021a) has implemented highly optimized CUDA kernels for transformer inference. To make further optimization, we design our custom kernels with the considerations below:

• Because padding tokens do not affect the final results, preventing padding tokens from participating in MHA instead of simply applying padding masks can significantly reduce the computational overhead.

• Although an attention mask is essential for MHA in text generation, constructing a mask that varies with the input length is time-consuming.

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

To remove previously mentioned masks in computation, we redesign the kernels and call 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 call the method as *thread block folding*. Next, we describe these two methods in detail.

2.1 Mask Fusion

The attention mask indicates the attention boundary for each token to prevent the attention from looking forward. The padding mask indicates where the padding tokens are. Thus they both characterize the position information of the tokens in a sequence. Meanwhile, each CUDA thread also has a unique positional index. So we can map each token in the MHA to a thread or block in the CUDA kernels. The function of the attention mask is achieved by comparing whether the CUDA position of the query token being processed is larger than the CUDA position of the key token. The function of the padding mask is achieved by starting the valid calculations from the padding offset when sequentially processing each token. Therefore, we transform the mask computation to logical operation with CUDA thread index comparison. Thus there is no need to store any explicit functional parameters of the masks and the computation overhead of masking operation is saved. The algorithm pseudo-code is shown in Algorithm 1.

Algorithm 1 MHA with *mask fusion*

Input: $qk, paddingLen, seqLen, batch, headNum$
Output: the attention weights back to qk
CUDA Initialize $grid \leftarrow (batch * headNum)$
CUDA Initialize $block \leftarrow (seqLen)$
$batchId \leftarrow blockIdx.x / headNum$
$padLen \leftarrow paddingLen[batchId]$
$qkOffset \leftarrow blockIdx.x * seqLen * seqLen$
$qkOffset \leftarrow qkOffset + paddLen * seqLen$
$s \leftarrow padLen$ ▷ start at first non-pad
$e \leftarrow seqLen$ ▷ end at last token
$reduceMax \leftarrow -inf$
$reduceSum \leftarrow 0$
for $i = s$ **to** e **do**
  $position \leftarrow qkOffset + threadIdx.x$
  $data \leftarrow qk[position]$
  $u \leftarrow padLen$ ▷ upper boundary
  $l \leftarrow i$ ▷ lower boundary
  if $l < threadIdx.x < u$ **then**
    $reduceMax \leftarrow blockReduceMax(data)$
    $reduceSum \leftarrow blockReduceSum(data)$
    $data \leftarrow softmax(reduceMax, reduceSum)$
  end if
  $qk[position] \leftarrow data$
end for

2.2 Thread Block Folding

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 an input length of 2048. However, the CUDA block only supports a maximum thread number of 1024, most inference frameworks, such as FT(v3.1) and LightSeq, have implemented kernels that restrict the model size and input length up to 1024, leading to limited availability.

To deal with large model sizes and sequence lengths, we propose to use several blocks to simulate a large block, shown as Figure 1. Imagine a virtual block large enough to hold all the tasks, then we can fold it once to create two 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. Then, the large model sizes or input lengths can be handled correctly, and a new CUDA thread dimension is created to man-

[1] https://github.com/pytorch/fairseq

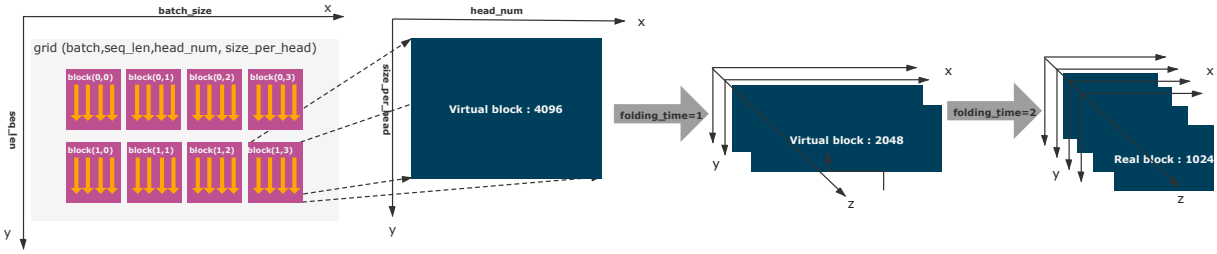

Figure 1: The schematic diagram of *thread block folding*.

age 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:

$$t = 2^{\lceil \frac{h}{1024} \rceil - 1}; \quad n = \frac{h}{2^t}$$

As for simplicity, *thread block folding* only adds a new dimension for the block, which slightly impacts the basic CUDA thread grid structure. 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 larger than 16384 and sequence length no longer than 4096.

3 Dynamic Memory Manager

The inference is much more sensitive to latency compared to training. As a result, model parallelism (Shoeybi et al., 2020) and pipeline parallelism (Huang et al., 2019) are undesirable for inference. Their communication overhead introduced by tensor slicing or layer split is significant even with the support of NVLink and GPUDirect. To reduce the latency and hardware requirements for online service, minimizing the memory footprint is of exceptional value when loading very large models. Thus we propose a dynamic memory management strategy for this issue.

Except for the model weights, the memory footprint includes the caches and the buffers. It is hard to reduce the memory footprint of weights because they are inherent to the model. Similarly, The K/V caches for MHA are also hard to compress because they are pre-allocated to avoid runtime memory requests, the size of which depends 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 operation result buffers.

3.1 Cache Reuse

The caches include K/V caches and activation caches. In incremental decoding, the keys and the values for every step are stored for the next step's attention computation. 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 required memory in advance to reduce the forward latency, avoiding malloc overhead and memory corruption.

Different from K/V caches, the activation results are useless after we have calculated and passed them to the next layer. The memory for these activations can be reused across different layers and different operators. We could reuse the activation caches in the following cases.

- The embedding operator shares the cache with the feed-forward operator and the final output. Yet the attention operator holds another cache because of the residual connection.
- The cache for input sequences can be reused by the decoded tokens. The maximum size is determined by the maximum input length.
- 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:

$$2 * b * h * p$$

The total K/V cache size is :

$$2 * b * h * s * l$$

3.2 Buffer Reuse

The continuous CUDA kernels are not always fused, especially when it comes to Cublas GEMM calls. So we need some buffers to store the returns for those non-fused kernels. Managing the buffers manually like FT is complicated and inefficient. We develop a dynamic buffer manager to avoid the tedium of manual design and achieve a highly efficient memory allocation.

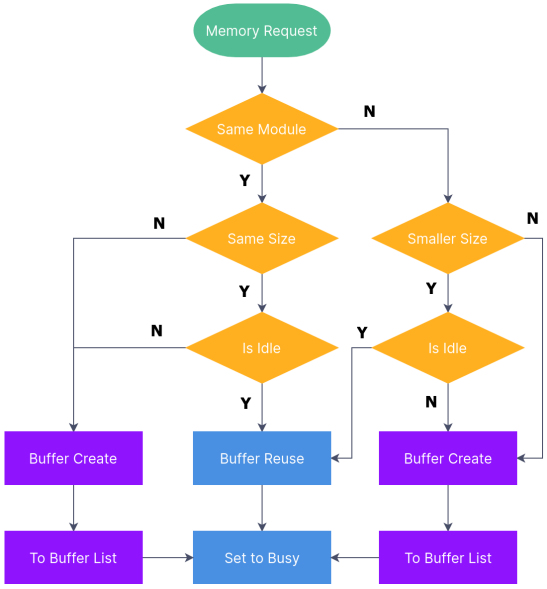

Figure 2: The schematic diagram of buffer decision strategy.

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 demonstrated in Figure 2. In our design, the developer only needs to request a buffer of a specified size and mark it as idle when it is useless, without concerning how to reuse memory exactly. The total buffer size is:

$$b * p * (6 * h + n * p)$$

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

4 Experiments

During inference, many factors can affect the actual performance, including model size, prompt length, sequence length, padding ratio in a batch, and hardware feature. Completely traversing all combinations requires a huge amount of works. Because the dataset has no effect on the experiment results, we adopt the fake inputs for convenience. To compare fairly and reduce our works, we define some typical experiment settings. If there is no special instruction, the experiment is conducted based on Configuration A in Table 1. Fairseq is an intuitional baseline because it is implemented using pure PyTorch code.

Table 1: Configuration A and B

	CONFIG A	CONFIG B
BATCH SIZE	4	8
MODEL SIZE	1024	2048
MAX PROMPT	1024	1024
MAX SEQUENCE	1024	1024
DATATYPE	FP16	FP16

4.1 Speedup for GPT-2 Layer with Different Sequence Lengths

We first apply EET over GPT-2 on NVIDIA 2080ti and A100. Figure 3 and 4 reveal that EET achieves about 4.48-20.27x and 4.30-27.43x speedup than Fairseq and about 1.21-6.30x and 1.62-8.16x speedup than FT(v3.1), on 2080ti and A100 respectively. For Fairseq and FT(v3.1), the incremental decoding processes the input tokens one by one, while EET improves the tokens parallelism by processing input tokens all at once. As a result, the speedup grows with the increase of the input length.

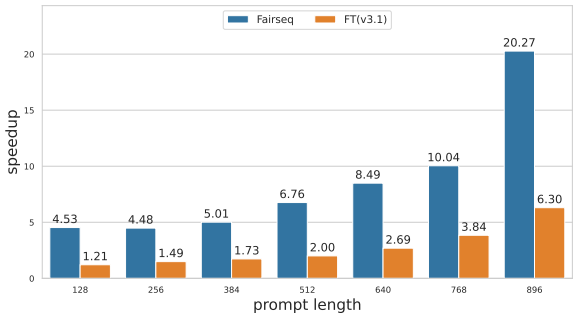

Figure 3: Inference speedup of EET with different prompt lengths on 2080ti compared to Fairseq and FT(v3.1).

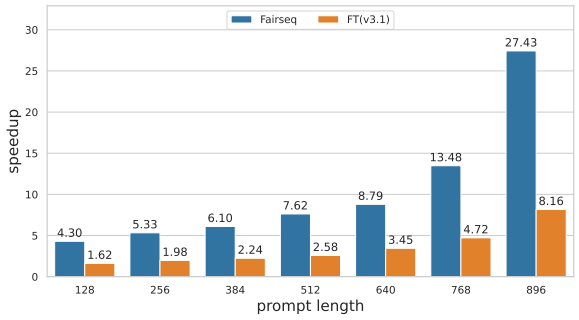

Figure 4: Inference speedup of EET with different prompt lengths on A100 compared to Fairseq and FT(v3.1).

The recent version of FT(v4.0) also introduces the parallel decoding of the input sequences for text generation as we did, so the performance of EET and FT(v4.0) is getting closer with the input length increasing. However, EET still has some performance advantages, which are attributed to our operation kernel optimization. Figure 5 shows that EET achieves about 1.40-2.54x speedup compared to FT(v4.0) with the configuration B in Table 1.

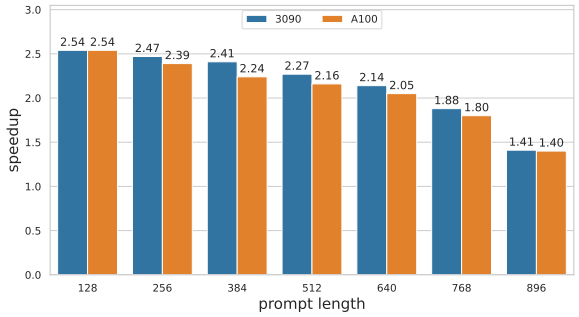

Figure 5: Inference speedup of EET with different prompt lengths on A100 and 3090 over FT(v4.0).

When processing a batch of inputs, the length of them may be uneven. The FT(v4.0) uses the minimum length of the prompts for full decoding, while the EET uses the maximum length. For example, if there is a batch containing sequences of different length like [5, 2, 4, 10], the final prompt length used for parallelism is 2 in the FT. In contrast, it is 10 in the EET. Figure 6 shows that we make 2.74-4.42x speedup with the prompt fixed to 512 and other configurations keeping the same as the configuration B in Table 1.

Unlike Fairseq and FT(v4.0), LightSeq only supports model sizes that are smaller than 1024, we also make a comparison here as a supplement. Figure 7 shows that we make 0.82-2.46x speedup when we set the model size to 768 and 1024.

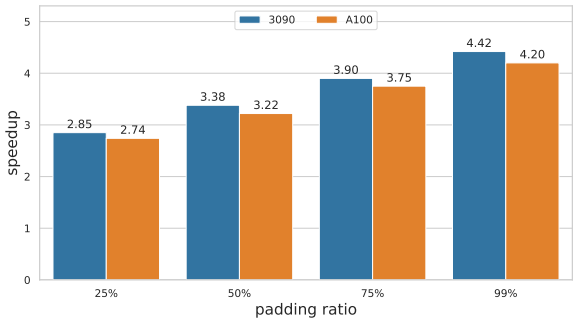

Figure 6: Inference speedup of EET with different padding ratio on A100 and 3090 compared to FT(v4.0)

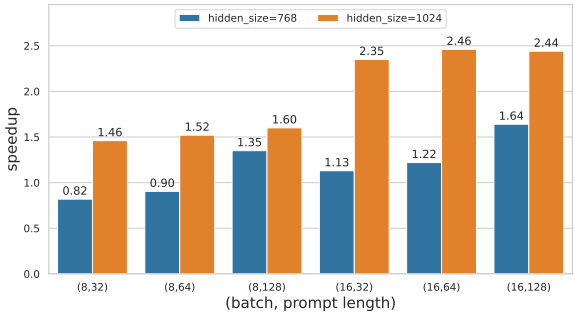

Figure 7: Speedup compared to LightSeq.

4.2 Speedup for Transformer Decoder Layer with Different Model Sizes

To prove the scalability of our EET, we evaluate the performance on different model sizes with configuration C in Table 2. Figure 8 and Figure 9 reveal that EET achieves about 2.25-7.50x speedup than Fairseq and about 1.71-4.61x speedup than FT(v4.0). The acceleration ratio decreases as the model size increases due to the increased ratio of matrix multiplication in the inference. Nevertheless, with the help of *thread block folding*, EET can still deliver significant speedup with very large model sizes, compared to Fairseq and FT(v4.0).

Table 2: Configuration C

	CONFIG C
BATCH SIZE	4 / 8
PROMPT	512
MAX SEQUENCE	1024
DATATYPE	FP16

4.3 Speedup for Bert Layer on 2080ti

We conduct experiments to validate the performance of the Bert encoder layer in EET on 2080ti. It is worth noting that the padding tokens take up

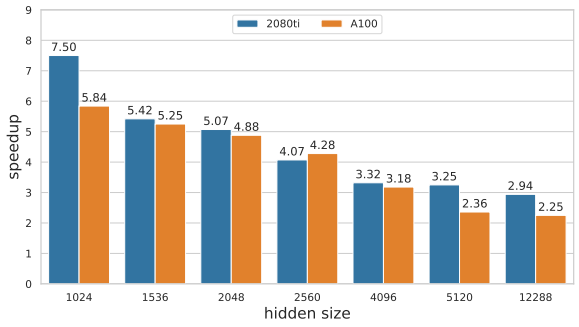

Figure 8: Speedup with different model sizes on 2080ti and A100 compared to Fairseq.

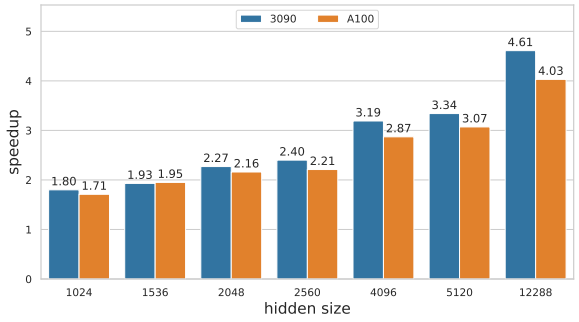

Figure 9: Speedup with different hidden sizes compared to FT(v4.0).

half of the total tokens. The result is shown in Figure 10. Deprecation of the padding masks with the *mask fusion* trick brings in 0.99-1.27x speedup. As for Bert, its hidden size is fixed to 1024 and it has no sequence mask, which kicks off the optimization of thread-block folding and sequence mask fusion, then the speedup is not as significant as GPT2.

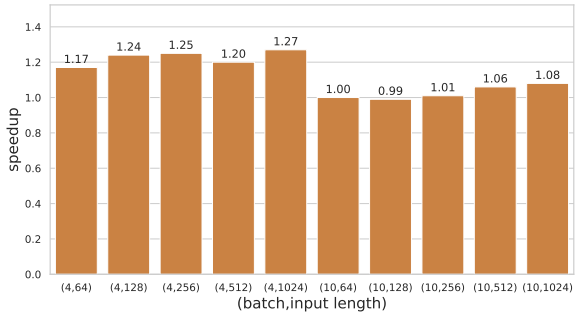

Figure 10: Performance speedup for Bert layer on 2080ti compared to FT(v4.0).

4.4 Memory distribution

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, as shown in Figure 11. Regardless of the hidden size, we can find that model weights and K/V caches occupy most memory. The activation caches and the buffers only take up a small part, which shows the effectiveness of our dynamic memory management strategy.

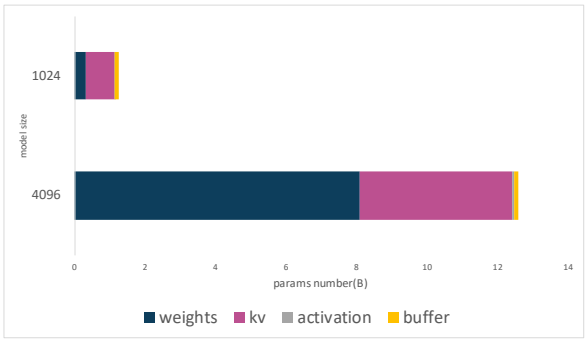

Figure 11: Memory distribution for 1024/4096 hidden sizes.

Given the batch size 4, the maximum sequence length 1024, we plot the memory occupancy of different model parameter sizes, see Figure 12. Compared with the 10 billion of PyTorch's maximum model parameter sizes, it is up to 18 billion for our EET, which proves that we can place much larger models onto one GPU, thus avoiding unnecessary waste of GPU resources and inter-GPU communication overhead on multiple cards.

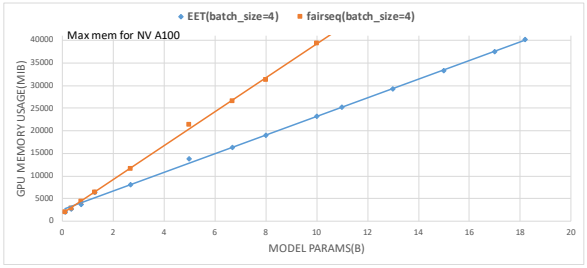

Figure 12: Memory occupancy for different model sizes.

5 Conclusion

This paper comprehensively describes a series of optimization techniques for transformer inference acceleration exploiting both algorithmic and GPU hardware features. These techniques are packed into the EET, a library dedicated to inference acceleration for large transformer-based models and long input lengths. EET has a 1.40-4.42x speedup for the GPT-2 layer and a 0.99-1.27x speedup for the Bert layer compared to the state-of-the-art transformer inference library FT. To make EET easier to

apply to a specific task, we provide operation level and model level API, meanwhile integrating web service with dynamic batching. We will continue to improve and keep it up-to-date.

Acknowledgments We would like to thank all the users of EET and the anonymous reviewers for their excellent feedback. This work is supported by the Key Research and Development Program of Zhejiang Province (No. 2022C01011).

References

Alexei Baevski, Wei-Ning Hsu, Alexis Conneau, and Michael Auli. 2021. Unsupervised speech recognition. *arXiv preprint arXiv:2105.11084*.

Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. 2020. wav2vec 2.0: A framework for self-supervised learning of speech representations. *arXiv preprint arXiv:2006.11477*.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2020. An image is worth 16x16 words: Transformers for image recognition at scale.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism.

Yifan Jiang, Shiyu Chang, and Zhangyang Wang. 2021. Transgan: Two transformers can make one strong gan.

Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020. Tinybert: Distilling bert for natural language understanding.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models.

NVIDIA. 2021a. "faster transformer".

NVIDIA. 2021b. "tensorrt".

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. 2019. Q-bert: Hessian based ultra low precision quantization of bert.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-lm: Training multi-billion parameter language models using model parallelism.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need.

Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. 2019. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned.

Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. 2021. Lightseq: A high performance inference library for transformers.

Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. 2021. Scaling vision transformers. *arXiv preprint arXiv:2106.04560*.