FastSeq: Make Sequence Generation Faster

Yu Yan[1] *, **Fei Hu**[1] *, **Jiusheng Chen**[1] *†, **Nikhil Bhendawade**[1] *, **Ting Ye**[1], **Yeyun Gong**[2], **Nan Duan**[2], **Desheng Cui**[1], **Bingyu Chi**[1] and **Ruifei Zhang**[1]
[1]Microsoft, [2] Microsoft Research Asia
{yyua,fhu,jiuchen,nibhenda,tiy, yegong,nanduan,decu,bingychi,bzhang}@microsoft.com

Abstract

Transformer-based models have made tremendous impacts in natural language generation. However the inference speed is a bottleneck due to large model size and intensive computing involved in auto-regressive decoding process. We develop FastSeq framework to accelerate sequence generation without accuracy loss. The proposed optimization techniques include an attention cache optimization, an efficient algorithm for detecting repeated n-grams, and an asynchronous generation pipeline with parallel I/O. These optimizations are general enough to be applicable to Transformer-based models (e.g., T5, GPT2, and UniLM). Our benchmark results on a set of widely used and diverse models demonstrate 4-9x inference speed gain. Additionally, FastSeq is easy to use with a simple one-line code change. The source code is available at https://github.com/microsoft/fastseq.

1 Introduction

Transformer-based model architectures have made tremendous impact in multiple domains. However, due to large model size and intensive computing involved in the decoding process, the inference speed is still a bottleneck for long sequences applications (Wu et al., 2016; Tay et al., 2020). A variety of model architectural innovations have been proposed to increase the generation speed from different perspectives. One trend is to change the model architectures, like model distillation (Shleifer and Rush, 2020) and sparse attention (Beltagy et al., 2020). Although these techniques can alleviate the performance issue, there may be still some trade-off between model accuracy and speed. On the other hand, efficient infrastructures have been developed to accelerate the inference speed, e.g., TensorRT (Vanholder, 2016) and FasterTransformers[1].

In this paper, we present FastSeq framework to make sequence generation faster. FastSeq can accelerate the sequence generation by 4x to 9x with a simple one-line code change for models in FairSeq (Ott et al., 2019) and Huggingface-Transformers (Wolf et al., 2020). The design principle of FastSeq is to improve the inference speed without losing model accuracy and usability.

Our optimization approaches include an attention cache optimization, an efficient algorithm for detecting repeated n-grams, and an asynchronous generation pipeline with parallel I/O. These optimizations are general enough for a wide range of Transformer-based model (Vaswani et al., 2017) architectures, including the encoder-decoder architecture (e.g., T5 Raffel et al. 2020, BART Lewis et al. 2020, ProphetNet Qi et al. 2020), the decoder-only architecture (e.g., GPT2 Radford et al. 2019), and the encoder-only architecture (e.g., UniLM Dong et al. 2019). FastSeq is also designed to be flexible for extension on supporting other models and frameworks. Our technologies are partially adopted by FairSeq[2]. A demo video can be found at https://www.youtube.com/watch?v=jrdsEUxhSEE.

2 Preliminary Analysis

For models with similar size, the sequence generation is much slower than classification, regression or language score computation. Why is the generation so time-consuming? Before analyzing the reasons, let's recap the generation algorithms first.

2.1 Generation Algorithms

Encoder-decoder structure is used in the most competitive models for sequence-to-sequence genera-

[*] Equal contribution

[†] Corresponding author

[1] FasterTransformer Github

[2] See pull requests FastSeq n-gram Blocking and Beam Search Perf Improvement

Proceedings of the Joint Conference of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: System Demonstrations, pages 218–226, August 1st - August 6th, 2021.

tion. The encoder side takes an input sequence of symbol representations $(x_1, ..., x_n)$ and outputs a sequence of continuous representations $\mathbf{z} = (z_1, ..., z_n)$. Then the decoder side generates an output sequence $(y_1, ..., y_t)$ with one element at a time. At each step, the model is auto-regressive by consuming the previously generated symbols and then computing the probability scores to select the next element. Greedy search and beam search are two popular algorithms used for the selection of next element. The difference between them is that at each step, greedy search only selects one candidate with maximum score, but beam search selects the top k candidates as beams. As beam search maintains multiple beams during the generation, it usually outputs a better result than greedy search.

To avoid repeated computation in the attention layer, the key (K) and value (V) from previous and current steps are usually cached to compute the next token. Equation (1) describes how the self-attention with the cache mechanism is implemented at step t.

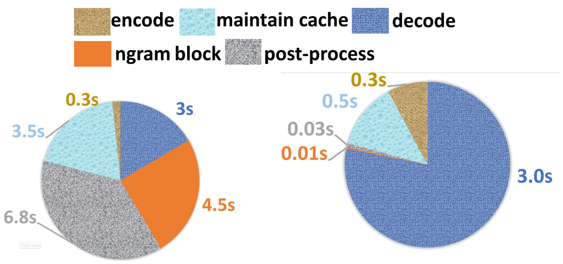
$$\begin{aligned}
\underset{[B\times M,1,D]}{Q_t} &= \underset{[B\times M,1,D]}{y_{t-1}} \cdot \underset{[D\times D]}{W_q} \\
\underset{[B\times M,t,D]}{K_t} &= concat(\underset{[B\times M,t-1,D]}{Cache\_K_{t-1}}, y_{t-1} \cdot \underset{[D\times D]}{W_k}) \\
\underset{[B\times M,t,D]}{V_t} &= concat(\underset{[B\times M,t-1,D]}{Cache\_V_{t-1}}, y_{t-1} \cdot \underset{[D\times D]}{W_v}) \\
\underset{[B\times M,1,D]}{attn_t} &= softmax(\frac{Q_t K_t^T}{\sqrt{d_{k_t}}})V_t
\end{aligned} \tag{1}$$

where B is the batch size; M is the beam size; D is the embedding dimension; Q_t, K_t, V_t represent **query**, **key**, **value** respectively, and are in the shape of $\mathbb{R}^{B\times M} \times \mathbb{R}^T \times \mathbb{R}^D$; W_q, W_k, W_v are the weights for the query, key, and value in the shape of $\mathbb{R}^{D\times D}$; $attn_t$ is in the shape of $\mathbb{R}^{B\times M} \times \mathbb{R}^1 \times \mathbb{R}^D$.

To simplify the equations, we do not consider multi-heads here, but these equations can be adjusted to be of multi-head style.

2.2 Bottlenecks in Generation

Figure 1a shows the profiling results of running the official BART model implemented by FairSeq. It indicates that maintaining cache, blocking n-gram repeats, and post-process individually take longer time than decoding itself. Profiling is done by running the official BART implemented by FairSeq v0.0.9 on CNN DM dataset with default parameters (batch size 32, beam size 4, and no-repeat n-gram 3). Non-computation parts, like maintain cache, blocking n-gram repeats and post-process, cost more than 80% of the generation time. We analyze these time-consuming components below.



(a) Before optimizations (b) After optimizations

Figure 1: (a) Before optimizations: non-computation operations, e.g, maintain cache, n-gram blocking and post-process cost most of the time. (b) After optimizations: majority of time is spent on encode and decode.

Cache Maintenance Along with better generation results, beam search introduces significant additional computational and memory cost. As Equation (1) indicates, the size of X_t, Q_t, K_t, V_t, and $attn_t$ in beam search is M times larger than those in greedy search. It results in more memory consumption, larger matrix operations (e.g., *concat*), and more expensive cache maintenance (e.g., reordering the top-k beams and the cached key and value at each step). Moreover, the batch size is constrained by large occupied memory, which results in a low GPU utilization.

Block N-Gram Repeats Blocking N-Gram Repeats is a widely used operation to avoid an n-gram appears more than once in natural language model (Paulus et al., 2018; Klein et al., 2017). It prohibits the repetitive generation of n-grams by setting their probability scores to zero. However, conventional implementation often needs to scan text sequentially and move data between GPU and CPU frequently. Its time complexity is quadratic in terms of sequence length. When processing long sequences, this operation becomes another bottleneck.

Post-process It deals with detokenization and final result output. Post-process performance is largely restricted by two parts: frequent exchange of small data between GPU and CPU and the detokenization efficiency. In addition, for a synchronized pipeline, post-process will block the generation for the next batch of samples, while there is no required dependency between these two components.

3 Design

In order to address above bottlenecks, optimizations need to be done at multiple levels, including operations, models, and pipelines, which basically touch every component of a sequence generation framework. It is a non-trivial burden for researchers and practitioners. As a result, we develop this FastSeq library to address these barriers and speed up end-to-end inference in sequence generation. FastSeq is designed with following features: (i) speed up the inference of sequence models without any accuracy loss; (ii) easy to use and compatible Python APIs with FairSeq and HuggingFace-Transformers; (iii) flexible to be extended to support new models and frameworks.

FastSeq is written in PyTorch (Paszke et al., 2019) and composed of (1) **ops** module: provide efficient implementations of kernels (e.g., block n-gram repeats); (2) **optimizer** module: optimize model implementations in run-time, where more efficient implementations will be automatically patched to replace the ones in existing NLP toolkits (e.g., FairSeq and HuggingFace-Transformers) or the deep learning libraries (e.g., PyTorch); (3) **models** module: define the model architectures (e.g., ProphetNet, UniLM). It is noteworthy that the models in FairSeq and HuggingFace-Transformers are natively supported as well. Only one-line code change is needed to make them work with FastSeq; (4) **command line interfaces (CLIs)** module: run the inference via commands with an asynchronous pipeline, including preprocess (e.g., tokenization), generation process, and post-process (e.g., detokenization). These CLIs are compatible with FairSeq and HuggingFace-Transformers as well. Users can use the same parameters to run their end-to-end inferences.

FastSeq is designed to be easy to use. Existing model usages (e.g., model content and parameter settings) in FairSeq and Huggingface-Transformers do not need to be changed. The example code can be found in below:

- Python API

```
# simply add the import of FastSeq

import fastseq
import torch

bart = torch.hub.load(
    'pytorch/fairseq',
    'bart.large.cnn')
bart.cuda().eval().half()
slines = [
    "Welcome to FastSeq. "
    "Hope you enjoy it."]
hypotheses = bart.sample(
    slines,
    beam=4,
    lenpen=2.0,
    max_len_b=140,
    min_len=55,
    no_repeat_ngram_size=3)
print(hypotheses)
```

- Command Line Interface

```
fastseq-generate-for-fairseq \
      DATA \
      --path MODEL \
      --fp16 \
      --task translation \
      --batch-size BATCH_SIZE \
      --gen-subset valid \
      --bpe gpt2 \
      --beam 4 \
      ...
```

4 Optimizations

To address the bottlenecks discovered in Section 2.2, we develop following optimizations.

4.1 Attention Cache Optimization

This section introduces how the cache for the key and value in self-attention and encoder-decoder attention can be optimized to further speed up the inference. We describe the cache deduplication below, see more comprehensive analysis and a new attention method with faster speed in our work EL-Attention (Yan et al., 2021)

4.1.1 Cache Optimization in Self-Attention

For the decoder-only or encoder-only Transformer models (e.g., GPT2, UniLM), X is the prefix of the generated hypothesis. In conventional implementations, X is replicated along beam dimension, and the corresponding partial in the key (K) and value (V) is same for each beam. This means, assuming K_t and V_t to be of dimension $[B, M, N + T, D]$, $K_0(b, i, n, d) = \cdots = K_t(b, j, n, d)$ and $V_0(b, i, n, d) = \cdots = V_t(b, j, n, d)$, for $\forall b \in [0, B)$, $\forall i, j \in [0, M)$, $\forall n \in [0, N)$, $\forall d \in [0, D)$, where N is the length of X, B is the batch size, M is the beam size, D is the embedding dimension.

To optimize the cache in self-attention, we can split the cached key and value in Equation (1) in two parts: $Cache\_K'$ and $Cache\_V'$ for the prefix; $Cache\_K_t$ and $Cache\_V_t$ for the generated sequence up till the time step t. With this split, the size of $Cache\_K'$ and $Cache\_V'$ can be reduced from $B \times M \times N \times D$ to $B \times 1 \times N \times D$. This also helps decrease cache reorder complexity by a factor of M.

However, the above split operation results in incompatible shapes between $Cache\_K'$ and $Cache\_K_t$, and between $Cache\_V'$ and $Cache\_V_t$. Instead of reshaping these cached keys and values, $einsum$ is utilized to compute $attn_t$. This way, the expensive $concat$ operations on large tensors can be avoided.

With the above changes, the matrix operations will be conducted on the tensors with much smaller size, so the peak memory can be smaller, the operations can run faster, and then a larger batch size can be leveraged. For example, at the step t, the sizes of $Cache\_K_{t-1}$ and $Cache\_V_{t-1}$ decrease from $B \times M \times (N+t-1) \times D$ to $B \times M \times (t-1) \times D$ by $\frac{N+t-1}{t-1}$ times. Then $concat(Cache\_K_{t-1}, x_t \cdot W_k)$ and $concat(Cache\_V_{t-1}, x_t \cdot W_V)$ can be much quicker than before due to less GPU memory allocation, copy, and deallocation. The peak memory during $concat$ is largely reduced as well. Meanwhile, this implementation will save the same amount of data movement when reordering the beams in $Cache\_K_{t-1}$ and $Cache\_V_{t-1}$ because $Cache\_K'$ and $Cache\_V'$ do not need to be frequently reordered since they are de-duplicated along beam dimension.

4.1.2 Cache Optimization in Encoder-Decoder Attention

The cached key and value in the encoder-decoder attention also have duplication. The reason is that the key and value in the encoder-decoder attention are calculated based on the final output hidden state (S) from the encoder side. Accordingly, the elements of cached key and value at the beam dimension are the same. Therefore, the size of $Cache\_K$ and $Cache\_V$ can be reduced by M times, from $B \times M \times N \times D$ to $B \times 1 \times N \times D$. Then the optimization benefits mentioned in Section 4.1.1 can be achieved here as well, including peak memory reduction and larger batch size. Additionally, the cached key and value are not needed to be frequently reordered since the elements at the beam dimension are exactly the same.

Notably, the above proposed optimizations are general and can be applied to a variety of models with different architectures if they share following features: 1) attention-based architectures, including self-attention or encoder-decoder attention; 2) auto-regressive decoding based on beam search. These models could be classic Transformer-based encoder-decoder architectures (e.g., BART, ProphetNet, T5), Transformer-based decoder-only architectures (e.g, GPT2), or Transformer-based encoder-only architectures (e.g., UniLM).

The detailed implementations of the optimized self-attention and encoder-decoder attention is provided in the Appendix.

Algorithm 1 GPU version no-repeat-ngram algorithm with arguments - ngram length n, previously generated tokens $tokens$, current step token probability distribution $probs$.

```
function BLOCK($tokens, probs, n$)
    $nBlk = tokens.rows$
    $nThr = tokens.columns + 1 - n$
    $shMem = sizeof(tokens.row(0))$
    BAN $<<< nBlk, nThr, shMem >>>$
    $(tokens, probs, n)$

function BAN($tokens, probs, n$)
    $row = blockIdx.x$
    copy $row$-th row of $tokens$ from global
    mem to shared mem $shm$
    $col = threadIdx.x$
    $start = tokens.columns + 1 - n$
    for $i = 0$ to $n - 1$ do
        if $shm[col + i] \neq shm[start + i]$ then
            return
    $tokenToBan = shm[col + n - 1]$
    $probs[row, tokenToBan] = 0$
```

4.2 GPU-based Block N-Gram Repeats Algorithm

As observed in Figure 1a, the cost of block n-gram repeats algorithm is as high as 25% of generation time. To reduce the cost, a new GPU-based kernel (see Algorithm 1) is developed to leverage the power of parallel compute and achieves the following benefits: 1) avoiding data movement between GPU and CPU to alleviate the throughput bottleneck of PCIe bus interface. 2) scanning n-grams in parallel. Instead of sequentially scanning tokens for detecting repeated n-grams, they can be scanned in parallel using threads equal to the number of n-grams generated till the time step t. Furthermore, each sample in a batch can be processed in parallel using multiple thread-blocks. 3) using GPU shared memory for faster memory access.

Since each token needs to be read multiple times

Model	Architecture	Task	Baseline	FastSeq	Speedup
encoder-decoder architecture					
BART (Lewis et al., 2020)	12L-12L-1024	CNN/DailyMail	2.4	18.4	7.7x
DistilBART (Wolf et al.)	12L-6L-1024	CNN/DailyMail	3.4	18.5	5.4x
ProphetNet (Qi et al., 2020)	12L-12L-1024	CNN/DailyMail	2.8	10.7	3.8x
T5 (Raffel et al., 2020)	12L-12L-768	WMT16 EN-RO	8.7	31.3	4.3x
Transformer (Ott et al., 2018)	6L-6L-1024	WMT16 EN-DE	96.0	417.0	4.3x
decoder-only architecture					
GPT2 (Radford et al., 2019)	0L-12L-768	CNN/DailyMail	3.0	16.7	5.5x
encoder-only architecture					
UniLM (Dong et al., 2019)	12L-0L-768	CNN/DailyMail	1.7	16.4	9.6x

Table 1: Benchmark results on models of different architectures. Speed is measured by samples/s.

(equal to token length of n-gram), they are stored in shared memory instead of global memory for faster access. Jia et al. (2018) reports shared memory bandwidth for Volta V100 is 16x of global memory bandwidth. Although there are multiple ways to organize CUDA thread blocks, our approach is to assign each n-gram to a thread and each thread-block to handle a sequence stream. In this way, Block N-gram repeats is parallelized along horizontal and vertical dimensions of a batch.

4.3 Asynchronous Pipeline with Parallel I/O

As shown in Figure 1a, post-process takes significant time (6.8s) in the generation process. It is under-optimized in many existing seq2seq frameworks. One reason is that post-process is not a part of the training process, many efforts are spent on optimizing the training pipeline and the model structure rather than the generation speed. Another reason is, despite of works focusing on generation speed, like distilling model, the speed metric only covers the computation time but does not include the post-process part. For example, FairSeq does not consider the post-process time when it measures the speed. These biases result in a big overlooked speed-up opportunity.

To improve the efficiency of the pipeline, we develop an asynchronous pipeline with parallel I/O. Similar to pre-fetch technology which loads next batch of data to GPU while running inference on the current batch, we post-process the current batch in a background thread while running generation on the next batch.

5 Evaluation

In the benchmarks, FairSeq and HuggingFace-Transformers are used as the baseline to evaluate the performance. The selected models cover different kinds of architectures, including the encoder-decoder models (e.g., BART, DistilBART, T5, ProphetNet), the decoder-only models (e.g., GPT2), and the encoder-only models (e.g., UniLM). CNN / Daily Mail dataset (Hermann et al., 2015) and WMT'16 (Bojar et al., 2016) are used as the benchmark datasets. The benchmark experiments are split into two groups 1) HuggingFace-Transformers with/without FastSeq; 2) FairSeq with/without FastSeq. If both FairSeq and HuggingFace-Transformers have implemented the model, we choose the faster result as the baseline.

Hardware The experiments are conducted on a node with 1 GPU (NVIDIA Tesla V100 PCIe 16GB) and 24 cores CPU (Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz).

5.1 End-to-end Performance

The end-to-end benchmarks (including model loading, preprocess, model inference, and post-process) have been conducted to evaluate the performance. For each model, we use the same configuration except batch size. We search the largest batch size for each framework by doubling it per search run. Each experiment is executed 10 times and the average running time is computed as the final result. The speed number is measured in samples per second.

With the optimizations of FastSeq, the end-to-end performance yields a roughly 4x to 9x speedup, see Table 1 for more details[3]. In the baseline, for summarization dataset CNN/DailyMail, the speed of all models (e.g., BART, DistilBART, ProphetNet, GPT2, UniLM) is between 1.7 and 3.4 samples per second. Enabling FastSeq boosts the speed to

[3] The baseline for BART is FairSeq and the baseline for DistilBART is Huggingface Transformers.

Model	Batch size	Cache GB	Throughput samples/s
BART$_{\text{large}}$ no cache	32	0.0	1.8 (0.7x)
BART$_{\text{large}}$	32	6.3	2.4 (1.0x)
+Asynchronous pipeline	32	6.3	3.6 (1.5x)
+GPU n-gram block	32	6.3	5.6 (2.3x)
+Attention cache optimize	32	1.8	8.1 (3.3x)
+Larger batch	128	7.2	18.4 (7.7x)

Table 2: BART$_{\text{large}}$ is the official version from FairSeq. No cache: disable cache on FairSeq. Generation parameters: beam size = 4, no-repeat n-gram = 3. Data: CNN DM validation dataset. Cache size is estimated according to max input length 1024, output length 50.

more than 10 samples per second for all models studied here, and the BART model achieves 18.4 samples per second, which is 7.7 times speedup. On the two WMT16 translation datasets, FastSeq improves throughput by 4.3 times.

In following sections, we will present analyses on the three optimizations used in FastSeq.

5.2 Analysis of the Cache Optimization

To evaluate effect of the cache optimizations introduced in Section 4.1, Table 2 compares the results of not using cache, using conventional cache, and using the proposed optimized cache. Although the computing complexity is the same for both cache-based approaches, the proposed cache optimization approach reduces the usage of GPU memory by 3.5 times. Such smaller cache memory can speed up $concat$ operations and reduce the data movement during the beam reordering, and also allow a larger batch size. These advantages together increase generation throughput from 5.6 to 18.4 samples/s.

5.3 Analysis of Block N-Gram Repeats

To demonstrate the effectiveness of GPU kernel described in Section 4.2, the new method is compared with two other methods in Table 3: 1) the one implemented by FairSeq (called baseline). 2) a revised CPU-based kernel, which improves baseline by moving data from GPU to CPU before computing to avoid multiple data transfers (called CPU kernel). The time difference (4477.1 ms vs 584.9 ms) between baseline and CPU kernel indicates that data transfer optimization alone can speedup about 8x. Furthermore, the proposed GPU kernel, which avoids data transfer and uses parallel computation has about 75x speed gain compared to CPU kernel. As shown in Figure 1b, the computing time after optimization becomes quite small, from about 25% to 1% of the overall time.

Method	Time (ms)
baseline	4477.1
CPU kernel	584.9
GPU kernel	7.8

Table 3: Compare three implementations of no-repeat n-gram.

Model	With fp16	Baseline R-1/R-2/R-L	FastSeq R-1/R-2/R-L
UniLM$_{\text{large}}$[4]	✗	43.08/20.43/40.34	43.09/20.29/40.32
UniLM$_{\text{large}}$	✓	43.06/20.42/40.32	43.08/20.29/40.32
BART$_{\text{large}}$	✗	44.21/21.20/41.03	44.21/21.20/41.03
BART$_{\text{large}}$	✓	44.22/21.20/41.04	44.22/21.21/41.03
ProphetNet$_{\text{large}}$	✗	44.20/21.17/41.30	44.20/21.17/41.30
ProphetNet$_{\text{large}}$	✓	44.17/21.17/41.28	44.17/21.17/41.28

Table 4: Metrics (ROUGE-1, ROUGE-2, and ROUGE-L) on CNN/DailyMail test set.

5.4 Analysis of Asynchronous Pipeline with Parallel I/O

Table 2 measures the performances of the synchronized pipeline with single process implemented by FairSeq and the proposed asynchronous pipeline with parallel I/O in FastSeq. The throughput is increased from 2.4 samples/s to 3.6 samples/s (around 1.5x). The speedup comes from the better resource scheduling, where the asynchronous pipeline allows post-process to run in the background when running the model inference, and the support of multi-thread detokenization. As shown in Figure 1b, the post-process unique time is reduced from about 38% to 1% of the overall time.

5.5 Analysis of Generation Quality

All optimizations in FastSeq do not affect the model generation quality. As discussed in Section 4, the logic for detecting the repeated n-gram blocks is the same for the CPU-based and GPU-based kernels, and the asynchronous pipeline with Parallel I/O only optimizes the I/O efficiency, so these two optimizations do not change the model outputs in any fashion. For the attention cache optimization, it does not affect model outputs in theory. However, in practice, if using mix precision (e.g., floating point 16) for inference, there may be a few trivial differences in the outputs due to the numerical stability issue in GPU. Similar differences can be observed when changing batch size during floating point 16 inference. But if using floating point 32, the generated results are exactly

[4] The differences between the ROUGE scores for UniLM are due to the differences in the data preprocess and the implementations of length-penalty.

the same. That means the minor differences are not caused by the proposed cache optimization itself. In FastSeq, the unit tests have been developed to make sure the inference outputs are the same with and without FastSeq when using floating point 32. We also compare the output quality based on the CNN/DailyMail dataset (Table 4). The quite similar ROUGE scores demonstrate that FastSeq does not impact the model quality.

6 Related Work

A variety of efforts have been developed to improve the efficiency of Transformer models. From the perspective of model architectures, there are efforts on reducing attention matrix size by chunking input sequences into blocks (Beltagy et al., 2020), or using strided convolution over the keys and queries to compress memory (Liu* et al., 2018). Another kind of approaches focus on reducing model size and memory consumption by weight quantization (Zafrir et al., 2019), weight sharing (Dehghani et al., 2019), and weight pruning (Michel et al., 2019). Knowledge distillation is another popular approach (Hinton et al., 2015).

On the other hand, a dozen of innovations on infrastructure side have been conducted to speed up serving of Transformer models. The fused chains of basic operators in the attention layers have been widely adopted in many frameworks (e.g., Onnx Runtime [5], Deep Speed[6]). It is also performance critical to optimize data layout and movement among the connected operations (Ivanov et al., 2020). In situation of varied input lengths, TurboTransformers (Fang et al., 2021) is developed to better serve online models by using dynamic batch scheduler, more efficient memory allocation and deallocation algorithms. FasterTransformers[7] deeply optimizes kernels of encoder, decoder and beam search to better utilize computer power of Tensor Core.

7 Conclusion

In this work, we present FastSeq, which provides general solutions for speeding up the sequence generation without accuracy loss. The proposed optimizations include an attention cache optimization, an GPU-based n-grams blocking algorithm, and an asynchronous generation pipeline. In the future, we will support more models and explore more techniques to accelerate the generation speed.

[5] https://github.com/microsoft/onnxruntime

[6] https://www.deepspeed.ai

[7] FasterTransformer Github

References

Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*.

Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Aurélie Névéol, Mariana Neves, Martin Popel, Matt Post, Raphael Rubino, Carolina Scarton, Lucia Specia, Marco Turchi, Karin Verspoor, and Marcos Zampieri. 2016. Findings of the 2016 conference on machine translation. In *Proceedings of the First Conference on Machine Translation: Volume 2, Shared Task Papers*, pages 131–198, Berlin, Germany. Association for Computational Linguistics.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. 2019. Universal transformers. In *International Conference on Learning Representations*.

Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. 2019. Unified language model pre-training for natural language understanding and generation. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402.

Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching machines to read and comprehend. In *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2020. Data movement is all you need: A case study on optimizing transformers. *arXiv e-prints*, pages arXiv–2007.

Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*.

Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander Rush. 2017. OpenNMT: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, pages 67–72, Vancouver, Canada. Association for Computational Linguistics.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Peter J. Liu*, Mohammad Saleh*, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. 2018. Generating wikipedia by summarizing long sequences. In *International Conference on Learning Representations*.

Paul Michel, Omer Levy, and Graham Neubig. 2019. Are sixteen heads really better than one? In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, Minneapolis, Minnesota. Association for Computational Linguistics.

Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. 2018. Scaling neural machine translation. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 1–9, Brussels, Belgium. Association for Computational Linguistics.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Romain Paulus, Caiming Xiong, and Richard Socher. 2018. A deep reinforced model for abstractive summarization. In *International Conference on Learning Representations*.

Weizhen Qi, Yu Yan, Yeyun Gong, Dayiheng Liu, Nan Duan, Jiusheng Chen, Ruofei Zhang, and Ming Zhou. 2020. ProphetNet: Predicting future n-gram for sequence-to-SequencePre-training. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 2401–2410, Online. Association for Computational Linguistics.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

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.

Sam Shleifer and Alexander M Rush. 2020. Pre-trained summarization distillation. *arXiv preprint arXiv:2010.13002*.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*.

Han Vanholder. 2016. Efficient inference with tensorrt.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation.

Yu Yan, Jiusheng Chen, Weizhen Qi, Nikhil Bhendawade, Yeyun Gong, Nan Duan, and Ruofei Zhang. 2021. El-attention: Memory efficient lossless attention for generation. *arXiv preprint arXiv:2105.04779*.

Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. 2019. Q8bert: Quantized 8bit bert. *arXiv preprint arXiv:1910.06188*.

A Cache Optimization in Self-Attention

First, we can split the cached key and value to two parts: $Cache\_K'$ and $Cache\_V'$ are for the prefix; $Cache\_K_t$ and $Cache\_V_t$ are for the generated sequence at the t step as below:

$$
\begin{aligned}
\underset{[B,1,N,D]}{Cache\_K'} &= \underset{[B\times 1,N,D]}{X} W_k \\
\underset{[B,1,N,D]}{Cache\_V'} &= XW_v \\
\underset{[B\times M,t,D]}{K_t} &= concat(\underset{[B\times M,t-1,D]}{Cache\_K_{t-1}}, y_{t-1} \cdot \underset{[D,D]}{W_k}) \\
\underset{[B\times M,t,D]}{V_t} &= concat(\underset{[B\times M,t-1,D]}{Cache\_V_{t-1}}, y_{t-1} \cdot \underset{[D,D]}{W_v})
\end{aligned}
\tag{2}
$$

The above split operation results in incompatible shapes between $Cache\_K'$ and $Cache\_K_t$, and between $Cache\_V'$ and $Cache\_V_t$. Instead of reorganizing these cached keys and values, Equation (3) is leveraged to compute $attn_t$. By this way, the expensive $concat$ operations on large tensors can be avoided.

$$
\begin{aligned}
\underset{[B\times M,1,N]}{attn\_w_0} &= einsum(Q_t, Cache\_K') \\
\underset{[B\times M,1,t]}{attn\_w_1} &= Q_t \cdot K_t^T \\
\underset{[B\times M,1,N+t]}{attn\_w} &= concat(attn\_w_0, attn\_w_1) \\
\underset{[B\times M,1,N+t]}{attn\_prob} &= softmax(\frac{attn\_w}{\sqrt{d_{k_t}}}) \\
\underset{[B\times M,1,N]}{attn\_prob_0}, \underset{[B,M,1,t]}{attn\_prob_1} &= split(attn\_prob) \\
\underset{[B\times M,1,D]}{attn_{t0}} &= einsum(attn\_prob_0, Cache\_V') \\
\underset{[B\times M,1,D]}{attn_{t1}} &= attn\_prob_1 \cdot V_t \\
\underset{[B\times M,1,D]}{attn_t} &= attn_{t0} + attn_{t1}
\end{aligned}
\tag{3}
$$

B Cache Optimization in Encoder-Decoder Attention

The first step is to remove the duplication in Cache_K and Cache_V. For the incompatible shape between Q and Cache_K, $einsum$ is leveraged to avoid the reshape.

$$
\begin{aligned}
\underset{[B,1,N,D]}{Cache\_K} &= \underset{[B,1,N,D]}{S} \cdot W_k \\
\underset{[B,1,N,D]}{Cache\_V} &= S \cdot W_v \\
\underset{[B\times M,1,N]}{attn\_w} &= einsum(Q_t, Cache\_K) \\
\underset{[B\times M,1,N]}{attn\_prob_t} &= softmax(\frac{attn\_w}{\sqrt{d_{k_t}}}) \\
\underset{[B\times M,1,D]}{attn_t} &= einsum(attn\_prob_t, Cache\_V)
\end{aligned}
\tag{4}
$$

As such, the size of $Cache\_K$ and $Cache\_V$ can be reduced by M times from $B \times M \times N \times D$ to $B \times 1 \times N \times D$. Then the optimization benefits in self-attention can be achieved here as well, including peak memory reduction and larger batch size. Additionally, the cached key and value are not needed to be reordered since the elements at the beam dimension are exactly the same.