LightSeq2: Accelerated Training for Transformer-Based Models on GPUs

Xiaohui Wang1, Yang Wei1, Ying Xiong1, Guyue Huang2, Xian Qian3, Yufei Ding2, Mingxuan Wang1, Lei Li2*

1ByteDance AI Lab, Shanghai, China
2University of California, Santa Barbara, California, USA
3ByteDance AI Lab, Mountain View, USA

{wangxiaohui.neo, weiyang.god, xiongying.taka, qian.xian, wangmingxuan.89}@bytedance.com
guyue@ucsb.edu, {yufeiding, leili}@cs.ucsb.edu

Abstract—Transformer-based neural models are used in many AI applications. Training these models is expensive, as it takes huge GPU resources and long duration. It is challenging because typical data like sentences have variable lengths, and Transformer's computation patterns are more complex than convolutional neural networks. Existing systems either only focus on model inference or optimization for only BERT-like encoder models. In this paper, we present LightSeq2, a system to accelerate training for a general family of Transformer models on GPUs. We propose a series of GPU optimization techniques tailored to the specific computation flow and memory access patterns of Transformer models. LightSeq2 supports many model architectures, including BERT (encoder-only), GPT (decoder-only), Transformer (encoder-decoder), and vision Transformer. Our experiments for a variety of models and benchmarks show that LightSeq2 is consistently faster (1.4-3.5×) than previous systems on different GPUs. In particular, it gains 308% training speedup compared with existing systems on a large public machine translation benchmark (WMT14 English-German).

Index Terms—Transformer, GPU Acceleration, Training, Natural Language Processing, Computer Vision

I. INTRODUCTION

Deep learning has been a prevailing approach to artificial intelligence (AI). Among various deep models, Transformers [1] have become one dominant choice of model architecture in many AI tasks, including natural language processing (NLP), computer vision (CV), and automatic speech recognition (ASR) [2]–[6]. Variants of Transformer have been proven to achieve state-of-the-art accuracy in text classification, question answering, machine translation, and visual object recognition tasks [2], [7], [8]. Transformer models typically require large model size and training data to perform well. For example, a GPT-3 model requires 3.1 million hours of training on modern GPUs and it costs $4.6 million to complete a single trial [9]. Fig. 1 shows model sizes and estimated training costs for several popular Transformer models. The training cost increases roughly in proportion to the number of model parameters. With the ever-growing model size, it becomes expensive to train them. Accelerating the computation for Transformers in both training and inference is critical.

*Partial work was done while at ByteDance.

However, existing approaches for accelerating Transformer computation are limited to either inference-only or encoder-only models. LightSeq [10], TurboTransformers [11], and FasterTransformer1 are recent systems targeting the serving of Transformers, but they cannot support Transformer training. DeepSpeed provides optimized training for Transformer [12], but it only supports Transformer encoder layers (e.g., BERT [2]). Tasks like machine translation requires full encoder-decoder Transformer layers, criterion layers for calculating generation loss, shared embedding, etc. These involve more complex computation flow, as in the cross attention computation between decoder and encoder layers. Therefore, it is nontrivial to accelerate the training for full Transformers.

Other research on general computation acceleration for neural networks include automatic hardware-oriented compilation and quantized computation [13], [14]. However, automatic compilation only supports fixed-length input, and find it difficult to deal with a variable-length input such as natural sentences for Transformer. Reducing the precision by quantization is beneficial in terms of performance but they could also lead to accuracy decrease to a certain extent, and accuracy is crucial for model training.

Comparing to inference, there are several additional challenges to accelerate training for Transformers. First, in addition to forward operators in inference, a training framework needs high-performance kernels to calculate gradients of each layer.

Fig. 1. Model size (number of parameters) and training time on Tesla V100 for popular Transformer-based models.

https://github.com/NVIDIA/FasterTransformer

1
Table I

<table>
<thead>
<tr>
<th>Training Libraries</th>
<th>Embedding</th>
<th>Encoder</th>
<th>Decoder</th>
<th>Criterion</th>
<th>Trainer</th>
<th>Sequence Length</th>
<th>DL Frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeepSpeed</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>Multiples of 16</td>
<td>✗</td>
</tr>
<tr>
<td>LightSeq2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Arbitrary</td>
<td>✓</td>
</tr>
</tbody>
</table>

Comparing systems for accelerated Transformer training. Other systems only support Transformer inference are not included.

II. BACKGROUND

A. Transformer Models

The main idea of Transformer is using multi-head attention (MHA) to map tokens in a sequence to feature representations. The network can be an encoder, a decoder, or an encoder-decoder architecture. Fig. 2 illustrates an encoder-decoder Transformer for machine translation (MT). Given an input sentence in the source language (e.g., German), the model first breaks the input sequence into tokens, and convert tokens into embedding vectors. These embeddings are processed by encoder and decoder layers. Both the encoder and the decoder contain multiple layers of MHA. Each MHA unit computes the attentional weights of one token by calculating the similarity scores to all token embeddings from the previous layer. The current layer’s embedding is then calculated from a weighted sum of embeddings of the previous layer. The decoder has two differences. It calculates self-attention to only prefix tokens in the decoder side. It includes additional cross attention from decoder to encoder tokens. The output sequence is generated token by token where each token probability is calculated from Softmax layer.

The Transformer-based models have variable-length intermediate tensors, which may lead to frequent allocation and release of GPU memory. There are several existing methods to deal with this problem. TurboTransformers [11] uses a sequence-length-aware allocator in their inference library to maximize non-dependent GPU memory sharing and memory consumption reduction. However, this also increases the frequencies of...
allocating and releasing the GPU memory. Thus it will slow
down the model inference. LightSeq [10] allocates the maximal
GPU memory in advance to prevent dynamic allocation and
release during training.

B. Model Training

There are four stages during each iteration of data-parallel
training.

1) The model receives a batch of training data and then
performs forward propagation, obtaining the final loss.
2) The model performs backward propagation using the
loss calculated after forward propagation, generating the
gradients of all parameters.
3) Gradients are gathered from all devices and then the
averaged gradients are computed and broadcast to each
device. There are two major families to complete this
process, all-reduce [15] and Parameter Server (PS) [16].
4) All parameters in each device are updated using the
averaged gradients. Since the initial state and gradient of
the parameters on each device are the same, the parameters
remain the same after one updated stage.

The bottleneck of the first and second stages is mainly in
computing, which depends on fast CUDA kernels. However, the
last two stages require better memory management, which will
speed up the copies of parameters and gradients. Many works
devote to accelerating these four stages, such as DeepSpeed,
Apex, DDP [17], etc. However, there is no work to accelerate
the complete training process. Fig. 3 is the time cost of the
four training stages. It can be found that model computing
and parameter updates account for a large proportion. After
training with LightSeq2, the time of the three stages is greatly
reduced, especially the parameter updates.

Different from model inference which only has the forward
propagation stage, model training is more challenging. First,
model training requires higher accuracy. Otherwise, the error
will be magnified after constant parameter updates. Second,
model training requires better memory management due to the
need for maintaining gradients and activation checkpointing
used by backward propagation. Finally, model training requires
a fast trainer for parameter updates. However, for Transformer
models, the training needs not deal with incremental length in
auto regressive decoding, which is simpler than inference.

III. RELATED WORK

Many approaches have been proposed to speed up the
computation for deep neural models, which can be divided
into two categories, algorithm-specific and algorithm-agnostic.

Algorithm-specific methods accelerate the training process by
improving the model architectures [18]–[24], training strategies
[25]–[27], optimization algorithms [28], [29], data precision
[30]–[32], and others [33], [34]. Vyas et al. [18] use clustered
attention to group queries into clusters and compute attention
just for the centroids, resulting in a linear complexity model.
Gong et al. [26] propose a stacking algorithm to transfer
knowledge from a shallow model to a deep model, then
applies stacking progressively to accelerate BERT training.
Yang et al. [28] propose ADAHESSIAN, a second-order
stochastic optimization algorithm that dynamically incorporates
the curvature of the loss function via adaptively estimates of
the Hessian matrix. These techniques can speed up the model
training to a certain extent but may affect the model structure
and effect, so the universality is not good.
For algorithm-agnostic optimization, Apex\textsuperscript{2} developed many commonly used GPU kernels using C++ and CUDA, including LayerNorm, Softmax and Adam optimizer. It supports automatic mixed precision computation similar to earlier quantization idea \cite{14} and distributed training. Unlike previous works that change the training behavior, the engineering level optimization strictly follows the training algorithm and has no impact on anything other than speed. Our LightSeq2 further enhances the performance of trainer with memory-efficient mixed precision computation without sacrificing the accuracy.

DeepSpeed \cite{35} integrates these small kernels into Transformer encoders, which boosts the training throughput on a single GPU and scales well on multiple GPUs. However, DeepSpeed has several limitations which hinder its usage in NLP, CV, and ASR tasks, especially in sequence generation tasks. First, DeepSpeed only optimizes Transformer encoder, thus it is not suitable for tasks requiring decoding modules (e.g., machine translation). Second, DeepSpeed does not optimize the other module like embedding and criterion, which prevents it from achieving higher performance. Third, DeepSpeed requires that the input length be an integer multiple of 16 due to the implementations of some kernels, which introduces unnecessary padding and computation. In contrast, LightSeq2 supports the arbitrary shape of inputs. Fourth, DeepSpeed does not support TensorFlow, which is also widely used in practice.

LightSeq (version 1.2) \cite{10}, TurboTransformers (version 0.5) \cite{11}, and FasterTransformer (version 4.0)\textsuperscript{1} are recent systems targeting the serving of Transformers. All systems exploit manually written CUDA kernels for accelerated forward computation of layers in a Transformer. They also improve the serving throughput by enhanced batch decoding strategies on GPUs. However, neither supports Transformer training since there are additional backward computation which is more complex than the forward pass.

TVM \cite{13} is a compiler that searches for candidate optimizations in the neural network according to specific patterns and automatically merge operators. It exposes graph-level and operator-level optimizations to provide performance portability to deep learning models. However, it is difficult to apply to Transformer-based models due to the variable input lengths.

IV. THE LIGHTSEQ2 SYSTEM

This section will introduce the four techniques used in LightSeq2 in detail, including computational graph optimizations, dependent reduction rewriting, accelerated mixed-precision update for trainer, and dangling-tensor aware memory manager.

A. Computational Graph Optimizations

Deep learning frameworks usually use computational graphs to represent the programs. Nodes represent operations like addition and multiplication, and edges represent tensor data flowing between operations.

We replace straightforward fine-grained GPU kernel functions in PyTorch or TensorFlow implementations with coarse-grain fused ones. Our customized operators avoid high overhead introduced by a mass of kernel function launches and GPU memory I/O for intermediate results during forward/backward propagation.

1) Transformer Layers: To all types of Transformers, we design fused kernel operators for both encoder and decoder layers.

The computational graph of a Transformer has two types of operations. One involves GEMM, including linear transformation and scaled dot product. The other is non-GEMM, such as Dropout, LayerNorm, and Softmax. To ensure the flexibility to cover various variants of the transformer-based models, we focus on fusing non-GEMM kernels and directly use the GEMM implementations from cuBLAS. The non-GEMM operations can be further grouped into two categories. One is element-wise operation (e.g., Dropout, ReLU, Reshape and bias adding), and the other is reduction operations (e.g. LayerNorm and Softmax). Element-wise operations enable explicit parallelism and multi-kernel fusion. Reduction operations require careful synchronization between threads.

Fig. 4 shows optimized computational graph for Transformer. The yellow boxes represent GEMM kernels and blue boxes represent custom non-GEMM kernels. Adjacent fine-grained element-wise kernels are fused into one coarse-grained kernel, resulting in fewer kernel launches and intermediate results. For example, the last kernel of the self-attention layer implements formation and scaled dot product. The other is non-GEMM, such as Dropout, LayerNorm, and Softmax. To ensure the flexibility to cover various variants of the transformer-based models, we focus on fusing non-GEMM kernels and directly use the GEMM implementations from cuBLAS. The non-GEMM operations can be further grouped into two categories. One is element-wise operation (e.g., Dropout, ReLU, Reshape and bias adding), and the other is reduction operations (e.g. LayerNorm and Softmax). Element-wise operations enable explicit parallelism and multi-kernel fusion. Reduction operations require careful synchronization between threads.

2) Embedding Layer: The embedding layer is widely used in most deep learning tasks to obtain the distributed representation of one word or an image patch. Given a token embedding lookup table $E$ and positional embedding lookup table $P$, we can get the representation $y^{(w,p)}$ of one token with index $w$ and position $p$:

$$y^{(w,p)} = \text{Dropout}(s \cdot E_w + P_p),$$

where $s$ is the embedding scale.
Considering an input sentence $x$ with length $l$, let $m$ denotes the Dropout mask generated in the forward propagation. We can efficiently compute the gradient of token $w$ in the embedding table:

$$
\nabla E = s \cdot \sum_{0 \leq i \leq l \text{ and } x_i = w} m^{(i)} \odot \nabla y(x_i, i),
$$

where $\odot$ represents an element-wise product. We use $\nabla x$ to denote $\frac{\partial L}{\partial x}$, where $L$ is loss from criterion layer, and $x$ is any intermediate layer output. This means summing up all gradients of tokens $w$ in the different positions in the sentence, which can be implemented in parallel by atomicAdd operation in CUDA to avoid interference from other threads.

3) Criterion Layer: The criterion layer is used to compute the loss between the model output and the ground truth. The loss is cross entropy for machine translation. Let $h_i$, a vector of length $V$, denote the output of the decoder for one token, where $V$ is the vocabulary size, and $y$ denote the one-hot vector of length $V$ representing the ground truth. The prevalent cross-entropy loss with label smoothing can be formulated as:

$$
L(p, q) = -\sum \limits_i p_i \log(q_i),
$$

where $p = (1 - \alpha)y + \frac{\alpha}{V} \cdot 1$ and $q = \text{Softmax}(h_i)$. $0 \leq \alpha \leq 1$ is the smoothing parameter, and $\sum_i q_i = 1$.

By plugging $p$ and $q$ into the loss function and calculating the partial derivatives of $h_i$, we can get the gradient of decoder output as follows.

First, the gradient of Softmax function is:

$$
\frac{\partial q_i}{\partial h_j} = \begin{cases} -q_i q_j & i \neq j \\ q_i(1 - q_i) & i = j \end{cases}
$$

We derive the gradient to decoder output in two situations. If $i$ is equal to the ground truth token index $k$, the gradient as:

$$
\nabla h_i = \frac{\partial L}{\partial h_i} = -\alpha \sum j \neq k \frac{1}{q_j} \cdot \frac{\partial q_j}{\partial h_i} - (1 - \alpha + \frac{\alpha}{V}) \cdot \frac{1}{q_k} \cdot \frac{\partial q_k}{\partial h_i}
$$

$$
= q_i - \frac{\alpha}{V} + 1 + \alpha
$$

The gradient for other slots are:

$$
\nabla h_i = \frac{\partial L}{\partial h_i} = -\alpha \sum \frac{1}{q_j} \cdot \frac{\partial q_j}{\partial h_i} - (1 - \alpha + \frac{\alpha}{V}) \cdot \frac{1}{q_k} \cdot \frac{\partial q_k}{\partial h_i}
$$

$$
= -\alpha \sum \frac{1}{q_j} \cdot \frac{\partial q_j}{\partial h_i} - \frac{\alpha}{V} \cdot \frac{1}{q_i} \cdot \frac{\partial q_i}{\partial h_i}
$$

$$
- (1 - \alpha + \frac{\alpha}{V}) \cdot \frac{1}{q_k} \cdot \frac{\partial q_k}{\partial h_i} = q_i - \frac{\alpha}{V}
$$

Therefore the final gradient to the decoder output is:

$$
\nabla h_i = \begin{cases} q_i - \frac{\alpha}{V} + 1 + \alpha & \text{if token } i \text{ is the ground truth} \\
q_i - \frac{\alpha}{V} & \text{otherwise}
\end{cases}
$$

which is an element-wise kernel of $q$ and can be executed in parallel. Our Softmax operation will be introduced later. But we do not directly calculate it. Instead, we calculate logarithm of Softmax by adding additional logarithmic operations and bias adding for forward and backward calculation.

4) Layer-batched cross Attention: As shown in Fig. 5(a), considering the model has $n$ decoder layers, the context (vectors like keys and values) in cross attention of decoder layer is obtained by calculating the linear transformation of the encoder output:

$$
\begin{align*}
& \hat{y}_{i, \text{key}} = W_{i, \text{key}} \cdot \sigma(x) + b_{i, \text{key}} \\
& \hat{y}_{i, \text{value}} = W_{i, \text{value}} \cdot x + b_{i, \text{value}}
\end{align*}
$$

where $\hat{y}_{i, \text{key}}$ and $\hat{y}_{i, \text{value}}$ are the contexts in cross attention of $i$-th decoder layer. $W_{i, \text{key}}$ and $W_{i, \text{value}}$ are the parameter matrices. $b_{i, \text{key}}$ and $b_{i, \text{value}}$ are the bias vectors. $x$ is the output of the encoder. As shown in Fig. 5(b), we merged the computation of the context of cross attention to reduce the launch of kernels and improve the concurrency of matrix multiplication:

$$
[y_{i, \text{key}}, y_{i, \text{value}}] = [W_{\text{key}}, W_{\text{value}}] \cdot x + [b_{\text{key}}, b_{\text{value}}]
$$

Splitting $[y_{i, \text{key}}, y_{i, \text{value}}]$ once will obtain context of all decoder layers in forward propagation. For backward propagation, LightSeq2 computes the gradient of loss with respect to $x$ (encoder output) after finishing the backward propagation of 0-th decoder layer.

B. Dependent Reduction Rewriting

LightSeq2 optimizes two batch reduction operations traditionally taking a long time in the training, including LayerNorm and Softmax.

LayerNorm normalizes the inputs $x$ using:

$$
y_i = \frac{x_i - \mu(x)}{\sigma(x)} + b_i,
$$

where $\mu(x)$ and $\sigma(x)$ stand for the mean and standard variance of $x$ respectively. Both are batch reduction operations. While warp-level parallelism provided by CUDA allows inter-thread communication for batch reduction over $x$, we cannot merge these two reductions directly. Because calculating variance
Fig. 6. Mixed-precision trainer in LightSeq2. Orange and blue blocks represent the model parameters and gradients respectively. Dotted blocks represent symbolic tensor link which have no actual memory storage. LightSeq2 eliminates FP32 copies of parameters/gradients and directly updates the FP16 workspace on-the-fly conversion.

For numerical stability, especially for mixed-precision training, it takes three steps to avoid overflow:
1) Find the maximal element of \( x \), denoted as \( x' \).
2) Deduct \( x' \) from each element in \( x \) so that the exponential never overflows, and then calculate the partition function \( Z = \sum_j \exp(x_j - x') \).
3) Calculate Softmax using \( y_i = \exp(x_i - x') / Z \).

Both step 1 and 2 are reduction operations. The number of reductions and reduction dimension are quite diverse in the attention Softmax. For example, in common NLP tasks, the reduction dimension ranges from a few to thousands and the number of reductions ranges from thousands to millions. To address this challenge, we implement multiple templates suitable for diverse shapes and let key parameters (e.g., number of blocks, warps per block, reduce times per block) tunable. LightSeq2 runs templates and searches for parameters before training to determine the optimal configuration for each shape.

C. Accelerated Mixed-Precision Update for Trainer

Trainer include numerical update algorithm for parameters (e.g. Adam). In mixed precision training [36], parameters and gradients are in FP16 during forward and backward propagation. Since the update values, the product of learning rates and gradients, are often tiny, the trainer need to maintain FP32 copies of parameters and gradients to ensure accuracy. A straightforward system copies each piece of gradients/parameters in the model to/from its FP32 partner in one training step (Fig. 6(a)). The trainer kernel will load the FP32 gradient to update the FP32 parameters. This mechanism has two disadvantages:

1) Numerous pieces of gradients/parameters lead to multiple fast-returning GPU kernels like copying and updating, which reduce GPU utilization.
2) Redundant memory footprints are caused by the FP32 copy of gradients/parameters.

Our LightSeq2 alleviates them by the symbolic tensor linking and the on-the-fly conversion. During the initialization of the trainer, LightSeq2 copies all pieces of parameters/gradients into one tensor called workspace orderly (Fig. 6(b)). Then we reset and link them as fragments of workspace. During each training step, LightSeq2 only executes the trainer kernel once to update the workspace, which prevents launching huge amount of fragmented GPU kernels on every piece of parameters/gradients.

where \( \alpha \) and \( \beta \) are coefficients that can be solved in parallel:

\[
\alpha = \frac{|x_i - \mu(x)|\mu(x) - \sigma(x)^2}{m\sigma(x)^4} \quad \beta = \frac{\mu(x) - x_i}{m\sigma(x)^3}
\]

The two batch reductions \( \sum_j w_j \nabla y_j \) and \( \sum_j w_j y_j x_j \) can be executed in parallel.

As LayerNorm is sensitive to the precision of floating points, LightSeq2 stores half-precision float point (FP16) for parameters and casts them to single-precision float point (FP32) during computation to avoid additional I/O cost.

Softmax Attention in Transformer needs Softmax. The forward process of Softmax can be expressed as \( y_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \).
Softmax FW  Dropout FW  cuBLAS  
Softmax BW  Encoder  
LightSeq2 Kernel  C++ Operator  Python API  
CUB  
Hugging Face  
Model Translator  
LightSeq2 Model Zoo  BERT  
Encoder  
Decoder  
Embedding  
Softmax  
…  
Trainer  
GPT-2  
ViT  
…  

Our trainer kernel loads the FP16 parameters/gradient from 
workspace to register and converts it on-the-fly to FP32. Then 
the parameters on register will be updated as usual. Finally, 
the parameters will be converted on-the-fly to FP16 and saved 
to workspace. Accessing memory with FP16 instead of FP32 
reduces the data movement by half and avoids the FP32 copies 
of parameters and gradients.

The cooperation between the symbolic tensor linking and the 
on-the-fly conversion leads to both memory savings and latency 
reduction without reducing accuracy. The experimental result on 
Transformer-big model shows that the proposed trainer reduces 
its memory usage by 2 GB and its runtime by 54.9% compared 
to the Fairseq trainer with high kernel fusion from Apex. Fig. 
7 is a simple implementation of the two technologies.

D. Dangling-Tensor Aware Memory Manager

In practice, training with a large batch contributes to fast 
convergence and higher accuracy. However, large batch training 
requires multiple GPU, gradient accumulation, or memory 
offload due to the memory limit.

LightSeq2 reduces the memory footprint by compacting 
memory with fewer allocation-and-releases at no extra cost. 
LightSeq2 divides the GPU memory into permanent memory 
with a fixed size to store parameters and gradients, and 
temporary memory with variable sizes to store intermediate 
tensors. To avoid the frequent allocation of temporary memory, 
we scan the training set and estimate the upper bound of the 
capacity. Thus temporary memory was allocated once with 
the maximal size before training starts, and it is reused for 
different batches and released after training finishes.

Fig. 8 shows LightSeq2’s temporary memory cost in the self-
attention backward process. The left side describes the steps of 
backpropagation. Each row on the right side lists the memory 
occupancy of temporary tensors in one step. Tensors in the 
same column reuse the same memory block. The sizes of the 
orange and purple tensors are $BLH$ and $BL^2N$. The tensors 
in the dashed memory blocks are not updated in that step, and 
those in the solid blocks are updated. We can see that only 
$3BH + \max\{3BH, BL^2N\}$ (the last block) bytes of memory are required, where $B, H, L, N$ 
denote the batch size, hidden dimension, max sequence length 
and the number of heads respectively. In contrast, without using 
the shared memory block strategy, a total of $9BLH + BL^2N$ 
bytes of memory will be required.

V. USABILITY DESIGN

We design LightSeq2 to be modular, usable and interoperable.

A. Software Architecture

The software architecture of LightSeq2 consists of five 
modules, as shown in Fig. 9. The lower two are internal and 
the upper three are exposed to users.

The CUDA kernel module implements efficient CUDA kernel 
operations, such as Softmax forward, Softmax backward, etc. 
These kernels are optimized for various input shapes. LightSeq2 
directly calls the external cuBLAS library for other kernels 
lke matrix multiplication (GEMM).

The C++ operator module encapsulates common operators in 
C++ using above CUDA kernels to includes both forward and 
backward procedures for encoders and decoders. In addition, 
LightSeq2 also utilizes GPU memory reusing technology to 
save GPU memory.

The Python API module provides many convenient APIs 
in Python by encapsulating C++ operators. With these APIs, 
Transformer model training can be implemented through just 
a few lines of codes. The Python API supports models trained 
with both PyTorch and TensorFlow.

The model zoo module implements many common models 
using Python APIs above. LightSeq2 provides a model transla-
from lightseq.training import LSTransformerEncoderLayer
config = LSTransformerEncoderLayer.get_config(
    model="bert-base",
    max_batch_tokens=4096,
    max_seq_len=512,
    fp16=True,
    local_rank=0)
ls_layer = LSTransformerEncoderLayer(config)
# replace the 1st Hugging Face layer with LightSeq2
bert_model.layer[0] = ls_layer

<table>
<thead>
<tr>
<th>Batch token size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>1.0</td>
</tr>
<tr>
<td>1024</td>
<td>1.2</td>
</tr>
<tr>
<td>2048</td>
<td>1.4</td>
</tr>
<tr>
<td>4096</td>
<td>1.6</td>
</tr>
<tr>
<td>8192</td>
<td>1.8</td>
</tr>
<tr>
<td>15000</td>
<td>2.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Batch token size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>1.0</td>
</tr>
<tr>
<td>1024</td>
<td>1.5</td>
</tr>
<tr>
<td>2048</td>
<td>2.0</td>
</tr>
<tr>
<td>4096</td>
<td>2.5</td>
</tr>
<tr>
<td>8192</td>
<td>3.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Batch token size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>1.0</td>
</tr>
<tr>
<td>1024</td>
<td>1.2</td>
</tr>
<tr>
<td>2048</td>
<td>1.5</td>
</tr>
<tr>
<td>4096</td>
<td>1.8</td>
</tr>
<tr>
<td>8192</td>
<td>2.0</td>
</tr>
<tr>
<td>15000</td>
<td>2.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Batch token size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>1.0</td>
</tr>
<tr>
<td>1024</td>
<td>1.5</td>
</tr>
<tr>
<td>2048</td>
<td>2.0</td>
</tr>
<tr>
<td>4096</td>
<td>2.5</td>
</tr>
<tr>
<td>8192</td>
<td>3.0</td>
</tr>
<tr>
<td>15000</td>
<td>3.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Batch token size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>1.0</td>
</tr>
<tr>
<td>1024</td>
<td>1.2</td>
</tr>
<tr>
<td>2048</td>
<td>1.5</td>
</tr>
<tr>
<td>4096</td>
<td>1.8</td>
</tr>
<tr>
<td>8192</td>
<td>2.0</td>
</tr>
<tr>
<td>15000</td>
<td>2.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Batch token size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>1.0</td>
</tr>
<tr>
<td>1024</td>
<td>1.5</td>
</tr>
<tr>
<td>2048</td>
<td>2.0</td>
</tr>
<tr>
<td>4096</td>
<td>2.5</td>
</tr>
<tr>
<td>8192</td>
<td>3.0</td>
</tr>
<tr>
<td>15000</td>
<td>3.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Batch token size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>1.0</td>
</tr>
<tr>
<td>1024</td>
<td>1.2</td>
</tr>
<tr>
<td>2048</td>
<td>1.4</td>
</tr>
<tr>
<td>4096</td>
<td>1.6</td>
</tr>
<tr>
<td>8192</td>
<td>1.8</td>
</tr>
<tr>
<td>15000</td>
<td>2.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Batch token size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>1.0</td>
</tr>
<tr>
<td>1024</td>
<td>1.5</td>
</tr>
<tr>
<td>2048</td>
<td>2.0</td>
</tr>
<tr>
<td>4096</td>
<td>2.5</td>
</tr>
<tr>
<td>8192</td>
<td>3.0</td>
</tr>
<tr>
<td>15000</td>
<td>3.0</td>
</tr>
</tbody>
</table>

1) Does LightSeq2 obtain consistent overall speedup for a variety of models across different GPUs?
2) How much benefit does each optimization technique bring?
3) How much is the memory saving in LightSeq2?
4) How does LightSeq2 scale with respect to model sizes and the number of GPUs?

A. Overall Speedup

1) Setup: Benchmarks: We include four tasks in the experiments (Table II): machine translation using WMT14 English-German data [37], image classification using CIFAR-10 data [38], text classification using GLUE data [39], and language modeling using WikiText data [40]. The models are Transformer (with full encoder-decoder), ViT, BERT and GPT-2. We evaluate each network architecture with different sizes and layer configurations to accommodate the need of a wide range of use cases.

Platforms: We run experiments using NVIDIA Tesla V100 and Tesla A100.

Baselines: We choose the state-of-the-art implementations for the benchmark tasks as the baselines: Fairseq [41], DeepSpeed [42], and Hugging Face [43].
### Hugging Face, Apex, and DeepSpeed

As shown in Table II, we use the out-of-the-box end-to-end implementation provided by these baselines and tune the setup based on our run-time environment. To measure the speed, we use words per second compared with Fairseq and samples per second compared with Hugging Face.

2) **Transformer:** We evaluate the performance of full Transformer on WMT14 English-German machine translation task. The baseline is Fairseq Transformer with Apex optimization. Apex can further improve the performance of Fairseq Transformer by providing custom optimized operators.

We train transformer models at three different sizes (e.g., 6e6d represents six encoder and decoder layers) on eight V100/A100 GPUs. The batch token size denotes the total number of tokens in one mini-batch. Fig. 10 show that LightSeq2 obtains a speedup of 1.4-2.8× on V100 and 1.5-3.5× on A100 compared to Fairseq Transformer with Apex optimization.

Due to the computational graph optimizations and dependent reductions rewriting, LightSeq2 achieves higher computational throughput and requires less GPU memory and I/O. As a result, LightSeq2 obtains a higher speedup on A100 than V100. Moreover, under scenario of training large model, the bottleneck is memory bandwidth rather than computational throughput due to the limited batch size. LightSeq2 also achieves higher speedup for larger model. To sum up, LightSeq2 favors GPUs with higher peak throughput and larger models.

3) **ViT:** We evaluate the performance of vision Transformer (ViT) [8] on image classification task using CIFAR-10 dataset.

4) **BERT:** We evaluate the performance of BERT (Transformer encoder only) on Microsoft Research Paraphrase Corpus (MRPC) task of General Language Understanding Evaluation (GLUE) benchmark.

The baseline is Hugging Face BERT and DeepSpeed BERT. Hugging Face implements BERT with native Pytorch operators and is widely used due to its public pre-trained models. DeepSpeed optimizes the Transformer encoder layer and

---

3 https://github.com/huggingface/transformers  
4 https://github.com/NVIDIA/apex  
5 https://github.com/microsoft/DeepSpeed  
6 https://github.com/microsoft/DeepSpeedExamples/tree/master/bing_bert
achieves the fastest performance for training BERT. For a fair comparison with DeepSpeed, we replace our efficient embedding, criterion, and trainer operators with theirs and individually verify the efficiency of our encoder layer.

We train BERT-Base and BERT-Large models on eight V100 GPUs and choose samples per second (SPS) as metric to evaluate performance. As shown in Fig. 13, LightSeq2 obtains a speedup of $1.44 \times$ for BERT-Base and $1.28 \times$ for BERT-Large compared to DeepSpeed, which proves the efficiency of the LightSeq2 encoder layer.

5) GPT-2: We evaluate the performance of GPT-2 (Transformer decoder only) on language modeling task using WikiText [42] dataset. Our baseline is Hugging Face GPT-2, which is based on native PyTorch operators.

We train GPT-2 Base with 117M parameters on eight V100 GPUs and GPT-2 Large with 762M parameters on eight A100 GPUs. Iterations per second is chosen as the metric to evaluate performance. As shown in Fig. 14, LightSeq2 obtains a speedup of $1.7-1.8 \times$ for GPT-2 Base on V100 and $1.6-1.9 \times$ for GPT-2 Large on A100 compared to Hugging Face, which proves the efficiency of the LightSeq2 decoder layer.

B. Speedup Breakdown

To demonstrate the benefits of our optimizations in detail, we test the performance of LightSeq2 from operator and layer level on one NVIDIA Tesla V100 GPU. To measure the speed, we run each operator/layer for 10 times and take the average time.

1) Operator-wise Speedup: We choose four common operations, Dropout (element-wise), Adam (element-wise), Softmax (reduction) and LayerNorm (reduction), to show the efficient of CUDA kernels from LightSeq2. Our baselines are their implementations from PyTorch, TensorFlow and DeepSpeed.

Fig. 16 shows the performance of LightSeq2 LayerNorm. LightSeq2 gains a speedup of about $4 \times$ despite of the batch token size and hidden dimension. However, with the increase of batch token size or hidden dimension, the speedup of DeepSpeed drops significantly. Our technique shows high performance even when the original kernel has limited parallelism. If the number of elements is huge, the speed of DeepSpeed is not even as good as PyTorch. On the other hand, TensorFlow is not as fast as PyTorch in most cases, except when there are too many elements.

Fig. 17(a) shows the performance of LightSeq2 Dropout. As the number of elements increases, both DeepSpeed and LightSeq2 become slower. When the number of elements is greater than five million, DeepSpeed becomes slower than PyTorch. The gap between TensorFlow and PyTorch becomes smaller, but LightSeq2 still has a speedup of $1.2-1.5 \times$.

Fig. 17(b) shows the performance of LightSeq2 Softmax.
Unlike the other kernels, as the batch size and sequence length increase, the speedup of LightSeq2 becomes larger, due to the specific optimization for different input shapes. The trends of DeepSpeed and TensorFlow are similar to the other kernels.

Fig. 18 shows the performance of LightSeq2 trainer. LightSeq2 gains a consistent speedup of $2.3 \times$ for Adam and $2.4 \times$ for SGD over Apex despite of model size, which demonstrates the benefits of the symbolic tensor link and on-the-fly conversion as discussed in Section IV-C.

Finally, as shown in Fig. 15, we evaluate the speed of only using kernel fusion or LightSeq2 trainer. As the batch token size increases, the speedup decreases due to the growing proportion of GEMM kernels. The gap between kernel fusion and LightSeq2 trainer becomes larger as the batch token size increases, which is consistent with Fig. 3. This is due to the fact that the forward and backward processes dominate as the batch token size grows.

2) Layer-wise Speedup: We evaluate the speedup of LightSeq2 layers over PyTorch implementations (Fairseq). The layer configurations are the same as in Transformer-Big models.

The results of forward and backward propagation are in Fig. 19. We can draw the following conclusions:

- LightSeq2 obtains higher speedup in forward propagation than in backward. This is because the time of backward propagation contains the part of gradient copies.
- The speedup ratios of the encoder and the decoder decrease rapidly as the sequence length becomes larger. However, the speedups of embedding and criterion are stable. This is because when the sequence length increases, the large matrix multiplication in the encoder and the decoder quickly saturates the GPU parallelism, leading to a sub-linear improvement.

In all cases, LightSeq2 layers are faster than PyTorch, which demonstrates the benefits of computational graph optimizations as discussed in Section IV-A.

C. Benefit of Memory Optimization

We evaluate the memory efficiency on WMT14 English-German machine translation task using V100 GPU. We use Fairseq as baseline, both using a batch token size of 8192. All experiments ran for 40 minutes to measure under a stable state.

Fig. 20 illustrates the GPU memory occupancy of both Transformer-Base (6e6d, 512d, 8 heads) and Transformer-Big (6e6d, 1024d, 16 heads) models. Fairseq consumes about 6 GB more GPU memory than LightSeq2 in both cases. For example, Transformer-Base models based on Fairseq can not run on a GPU with only 16 GB memory. Another phenomenon is that the GPU memory of Fairseq will gradually increase as it runs. This is because Fairseq dynamically allocates and releases the GPU memory when the sequence lengths differ. Thus, Fairseq needs to apply for additional GPU memory if a long sequence is input into the model. In contrast, LightSeq2 allocates the maximal GPU memory in advance, so there will be no memory change during the training. As shown in Fig. 10, LightSeq2 can train deep models under large batch size (batch size 15k for 6e6d Transformer and batch size 8192 for 12e12d Transformer on 32 GB V100), whereas Fairseq Transformer encounters out-of-memory error, which is useful for training deep models.

Fig. 21 illustrates the GPU utilization of both Transformer-Base and Transformer-Big models. In the whole training process, LightSeq2 keeps a utilization rate of about 99% in both cases. However, for Fairseq, the utilization of the Transformer-Base model is very unstable. The lowest is only 80%, and in most cases, it fluctuates between 87% and 93%, mainly due to frequent memory allocation and release. The utilization of the Transformer-Big model is much more stable, but the highest is only 95%.

If the batch token size is smaller, the gap between the two implementations will be more obvious. For example, when the batch token size is reduced to 4096, the memory utilization of the Transformer-Base model based on Fairseq is only 73%, while the LightSeq2 is as high as 96%.

D. Scalability

To evaluate the scalability of LightSeq2, we extend the experiments in Section VI-A2 to multiple nodes, each with 8 A100 GPUs. LightSeq2 provides specific optimizations for
the computation of Transformer and general optimizations for popular trainers such as Adam and SGD. We adopt the same all-reduce synchronization strategy from PyTorch. As the number of nodes increases, LightSeq2 obtains a speedup of $1.14-1.41 \times$ for 48e48d model on different numbers of GPUs (Fig. 22(a)) and $1.12-1.22 \times$ on 5 x 8 GPUs for different sizes of models (Fig. 22(b)). These results show that LightSeq2 scale well and gain consistent speedup for distributed training scenario and for increasing model sizes.

With the increase of number of GPUs or model sizes, the proportion of time for synchronization during training increases. As a result, the speedup decreases as number of GPUs or model size increases. Nevertheless, LightSeq2 still obtains a speedup of 1.12× on 5 x 8 A100 GPUs for a large Transformer model with a total of 120 layers (60e60d), which shows the system’s capability for large model on many GPUs. In the future, LightSeq2 can be further integrated with synchronization optimization techniques such as gradient compression, which are orthogonal to the focus of the current paper.

**VII. CONCLUSION**

In this paper, we describe a series of engineering-based GPU optimization techniques for fast Transformer training. Compared with existing approaches, our system strictly follows the standard training algorithm, therefore guarantees the quality and reproducibility of existing models. We systematically compare our work with existing state-of-the-art systems with various settings and analyze each component’s performance, demonstrating the solidness and scalability of the contribution. Compared to PyTorch and TensorFlow implementations, LightSeq2 can obtain a speedup of up to 3× under different configurations.

In the future, there are still room for additional optimization such as padding removing and better memory management strategy.

**ACKNOWLEDGMENT**

We would like to thank the colleagues in Volctrans team for their contributions in project development. We also thank Yaxin Ding for proof-reading, and the colleagues and classmates at University of California Santa Barbara for their suggestions on paper writing. GH, YD and LL’s work at UCSB are partially supported by UCSB-IEE grant from Meta. We would also like to thank the contributors and users for the open source project LightSeq.

**REFERENCES**


