TurboTransformers: An Efficient GPU Serving System For Transformer Models

Jiarui Fang
Pattern Recognition Center, Wechat AI, Tencent Inc
Beijing, China
jiaruifang@tencent.com

Yang Yu
Pattern Recognition Center, Wechat AI, Tencent Inc
Beijing, China
josephyu@tencent.com

Chengduo Zhao
Pattern Recognition Center, Wechat AI, Tencent Inc
Beijing, China
florianzhao@tencent.com

Jie Zhou
Pattern Recognition Center, Wechat AI, Tencent Inc
Beijing, China
withtomzhou@tencent.com

arXiv:2010.05680v4 [cs.DC] 20 Feb 2021

Abstract

The transformer is the most critical algorithm innovation of the Nature Language Processing (NLP) field in recent years. Unlike the Recurrent Neural Network (RNN) models, Transformers can process on dimensions of sequence lengths in parallel, therefore leading to better accuracy on long sequences. However, efficient deployments of them for online services in data centers equipped with GPUs are not easy. First, more computation introduced by transformer structures makes it more challenging to meet the latency and throughput constraints of serving. Second, NLP tasks take in sentences of variable length. The variability of input dimensions brings a severe problem to efficient memory management and serving optimization.

This paper designed a transformer serving system called TurboTransformers, which consists of a computing runtime and a serving framework to solve the above challenges. Three innovative features make it stand out from other similar works. An efficient parallel algorithm is proposed for GPU-based batch reduction operations, like Softmax and LayerNorm, major hot spots besides BLAS routines. A memory allocation algorithm, which better balances the memory footprint and allocation/free efficiency, is designed for variable-length input situations. A serving framework equipped with a new batch scheduler using dynamic programming achieves the optimal throughput on variable-length requests. The system can achieve the state-of-the-art transformer model serving performance on GPU platforms and can be seamlessly integrated into your PyTorch code with a few lines of code.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPoPP '21, February 27-March 3, 2021, Virtual Event, Republic of Korea
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441578

CCS Concepts: • **Computing methodologies** **Concurrent computing methodologies**; **Natural language generation**; **Parallel algorithms**.

Keywords: Transformers, Deep Learning Runtime, Serving System, GPU

1 Introduction

The recent success of Nature Language Processing (NLP) techniques is enabled largely by the transformer-based Deep Neural Networks (DNNs), such as Seq2seq [30], BERT [7], GPT2 [25], and XLNet [31], ALBERT [14]. With the support of the attention mechanism, the transformer models can capture long-range dependency in long sequences. In data centers, GPU has proven to be the most effective hardware for deploying deep learning (DL) services. The DL service accepts network requests and performs inference computation by performing a feed-forward pass on the model. Although there are mature solutions for Convolutional Neural Network (CNN) and RNN services, deploying transformer services with low latency and high throughput on GPU still faces two critical challenges.

Despite their success in model accuracy, transformer models are notorious for the massive amount of computation. For the inference of a 40 words sequence, the base BERT model requires 6.9 Gflops. To translate a 20 words sentence from Chinese to English, a typical Seq2seq model requires over 20 Gflops. In comparison, for the inference of a 3x224x224 image, the ResNet50 [11], GoogleNet [29] and AlexNet [13] require 3.9, 1.6 and 0.7 Gflops, respectively. Generally, transformer models lead to more computations than previous DNN models.

Besides enhanced computation requirement, transformer models introduced the problem of **variable-length** input, where intermediate tensor dimensions have to change according to the input sequence length during a serving process. Although also facing at variable-length input, RNN-based models, like LSTM [12] and GRU [4], split the variable-length input into multiple fixed-length inputs and execute them sequentially. Unlike models with fixed-length input,

transformer models cannot benefit from pre-optimization of memory space allocated for intermediate tensors with known lengths, resulting in memory optimization challenges. In addition, due to variable-length input, the optimization of the serving framework becomes more complicated. Conventional serving frameworks take advantage of batching techniques to increase GPU execution efficiency. Extra computations brought by zero-padding of short requests in a batch of variable-length requests often conflict with the performance gains of batching computing.

Existing online serving solutions are not able to resolve both the large computation demand and the variable-length input issues, especially the latter one. The deficiencies can be summarized from the three following aspects. First, directly employing a training framework, such as TensorFlow [1] or PyTorch [23], on inference tasks is not able to fully utilize the hardware resources. Inference differs from training in a way that it does not perform backward propagations which require extra memory space for intermediate tensors and eliminate the opportunity for operator fusion. Second, currently existing DL inference frameworks such as onnxruntime [19], TenorFlow XLA [10], TVM [3], tensorRT [22] use techniques designed for fixed-length input workloads and have insufficient capability in dealing with variable-length input. Most of these current frameworks need a time-consuming preprocessing step to tune the computation pattern of the model operators according to their pre-determined input dimension. In order to work with transformer models, they usually convert variable-length requests into fixed-length requests through zero paddings, which not only introduces additional computational overhead but also limits the maximum length allowed to the pre-determined value. Although onnxruntime [19] recently provides some patches to support computation for variable-length input, the runtime's GPU memory management is not efficient. After it serves a long request or a large batch of requests, a huge amount of memory allocated for intermediate tensors will not be released, introducing waste in terms of the memory footprint. Third, none of the existing solutions have investigated serving optimization for variable-length input workloads. The request batching, which is the technique most helpful for performance, adopted in modern serving systems, such as TF-serving [9], Clipper [5], Nexus [28], are suitable for only fixed-length input.

To solve the challenges of deploying an efficient transformer service, we proposed a serving system called Turbo-Transformers. The system consists of a light-weight computation runtime and a serving framework. The runtime adopts a variable-length-input-friendly design to avoid time-consuming preprocessing specified with dimensions of the intermediate tensor. After loading a pre-trained model, the runtime rewrites the computation graph by fusing non-GEMM kernels, and provides efficient CUDA implementations for them. Before launching an inference, it conducts light-weight memory usage optimizations according to the input sequence length. The runtime can achieve state-of-the-art speed and a smaller memory footprint compared with existing DL runtimes. Moreover, it is easy to use, It can bring end-to-end speedup by adding a few lines of Python code. Building upon the runtime, the serving framework improves throughput of the service through a variable-length-aware batching technique.

The innovative contributions of the paper are listed as follows.

- We proposed a new parallel algorithm for Batch Reduction kernels like Softmax and LayerNorm, which pushes the efficiency boundary of these kernels on GPU.
- We proposed a sequence-length-aware memory allocation algorithm. Unlike other popular allocators for DNN runtimes, our allocator can utilize the computation-graph of the DNN model to derive efficient memory reusing strategies for variable dimension intermediate tensors.
- We proposed a sequence-length-aware batch scheduler. It utilizes a dynamic programming algorithm to derive a batching scheme to achieve the optimal throughput.

2 Backgrounds

2.1 Transformer Models

Self-attention is the key idea behind the transformer model. It has the ability to attend to different positions of the input sequence to compute a representation of that sequence. A transformer model handles variable-length input using stacks of self-attention layers instead of RNNs or CNNs. An encoder-decoder model architecture is illustrated in Figure 1.

Multi-head attention consists of four parts, a set of linear layers that are split into heads, a scaled dot-product attention, a concat, and a final linear layer. Each multi-head attention block gets three tensors as inputs; Q (query), K (key), V (value). These are put through the linear layers and split up into multiple heads. The scaled dot-product attention computes the dot products of the query with all keys, and applies a Softmax function to obtain the weights on the values. The attention output for each head is then concatenated and put through a final linear layer. In addition to multi-head attention, each of the layers in our encoder and decoder contains a fully connected feed-forward network to improve the capacity of the model. It consists of two linear transformations with activations in between.

By introducing parallelism on the sequence length, dimensions of Q, K, and V tensors change unpredictably during serving. Take a real-time translation system as an example. After a short greeting phrase including a few words as input, a long paragraph of hundreds of words maybe its next input. The variable-length input feature brings difficulties to batching. When the input contains a batch of request, in order to be processed together, the short sequence must be filled with

zeros according to the longest sequence before being sent to the transformer models.

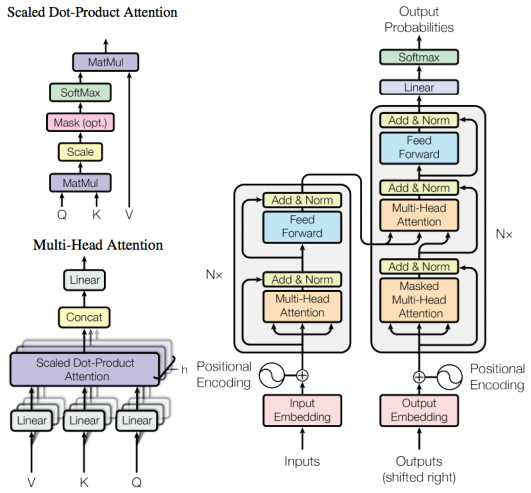

Figure 1. The transformer model [30] - an encoder-decoder model architecture and its key components.

Figure 1 shows a transformer architecture with both encoder and decoder parts. Note that decoder parts are not necessary in transformer-based model, for example the widely-used BERT model only contain the encoder parts.

2.2 Serving Systems

Leveraging the power of transformer-based models to NLP online applications requires joint efforts in both training and serving stages. Training is the process of building a model from offline data, which requires iteratively forward and backward passes. Serving, consisting of repeated inferences, is the process of using the model to extract useful features from user online input through a forward pass. Although inference does not involve complex backward computation and iterative processing, its performance requirements are more demanding. The serving process must run in real-time with low latency and sometimes should have the capacity of handling orders of magnitude more throughput than the training process. A serving system can be divided into two components, a DL inference runtime, and a serving framework.

Training frameworks have been widely used as inference runtimes. For instance, TF-serving [9] is wrapped with TensorFlow. Considering the poor performance of applying training framework in inference, some works have been dedicated to inference-specific runtimes, such as onnxruntime [19], TenorFlow XLA [10], TVM [3], and TensorRT [22]. Due to time-consuming preprocessing specific to the dimension of inputs, most of these runtimes can not be applied in variable-length input tasks. Among them, only the onnxruntime is able to be used in the variable-length input tasks, with dynamic axis supports after version 1.3. Faster Transformers [20] is a transformer boost software developed by NVIDIA. However, it is not a complete runtime because it has no memory manager and has to be used as an operator for TensorFlow. The comparison between this work and them are listed in Table 1.

Table 1. Comparison of our runtime and existing GPU DL Inference runtimes.

Related Works	Speed	Preprocess	Variable-Len	Usage
Tensorflow-XLA [10]	Medium	Yes	No	Easy
PyTorch [23]	Medium	No	Yes	Easy
TensorRT [22]	Fastest	Yes	No	Hard
Faster Transformers [20]	Fast	Yes	No	Hard
ONNX-runtime [19]	Fast	Yes	Yes	Medium
TurboTransformers	**Fastest**	**No**	**Yes**	**Easy**

The serving framework wraps the runtime into a service exposing gRPC/HTTP as endpoints. The advanced functionalities of the serving framework include batching, caching, model version management, and model ensembles. Batching boosts throughput substantially by combining multiple inference requests into a batch to increase GPU usability, which is the main focus of this paper. TF-serving enforces a static batch size by concat multiple requests together and has to pad zeros if requests are not enough. Clipper [5] proposed an adaptive batching scheme to dynamically find and adapt the maximum batch size. Nexus [28] further designed a batch scheduler to serve multiple different models on the same GPU. Ebird [6] is a prototype system designed an elastic batch scheduler based on an inference engine supporting multiple batches of the same model running concurrently. All of the above works are targeted at fixed-length input, which does not consider performance harm brought by zero-padding of short requests in a batch of variable-length requests. To avoid the zero-padding problem in RNN, BatchMaker [8] breaks the computation graph into a graph of cellulars and dynamically decides the set of cellulars should be batched together. It takes advantage of the weight sharing between multiple forward passes of RNN, which is not applicable in transformer models.

3 Design Overview

As shown in the Figure 2, there are two ingredients of TurboTransformers, an **inference runtime** and a **serving framework**. The system accepts network requests and responds the results processed by the Transformer models to end users. Section 4 will elaborate on the details of the runtime[1], and Section 5 will focus on the serving framework.

4 Inference Runtime

Inference runtime focuses on increasing computation efficiency and optimizing memory allocation.

[1]The code of runtime is publicly available at https://github.com/Tencent/TurboTransformers.

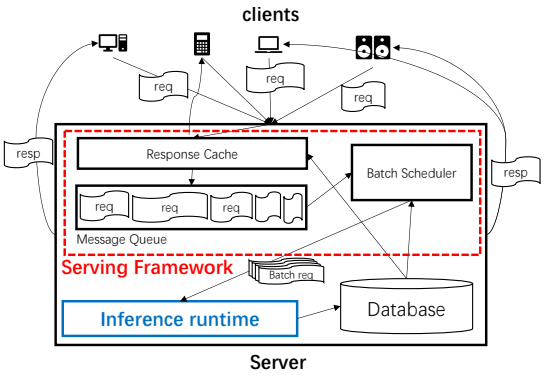

Figure 2. The serving system architecture adopted by Turbo-Transformers.

4.1 Computational Optimizations

4.1.1 Kernel Fusion. Without backward propagations, there is a lot of room left for inference customized optimizations. When using training frameworks, like TensorFlow or PyTorch to infer a transformer model on the GPU, a significant amount of time is spent on some non-computational intensive kernels. Take PyTorch as an example. For the case where batch size and sequence length are both relatively large, PyTorch BERT brings low efficiency due to the inefficient implementations of non-GEMM kernels. For a BERT inference on a Tesla V100 GPU using input with batch size as 20 and sequence length as 128, only 61.8% of the time is spent on GEMM kernels, and 38.2% is spent on non-GEMM intensive cores, such as LayerNorm, Softmax, Add Bias, Transpose, etc. For the case where batch size and sequence length are relatively small, PyTorch leads to poor efficiency due to the launch overhead of the CUDA kernels. For a BERT model inference on a Tesla V100 GPU with batch size as 1 and sequence length as 40, GPU is completely idle 80.64% of the time.

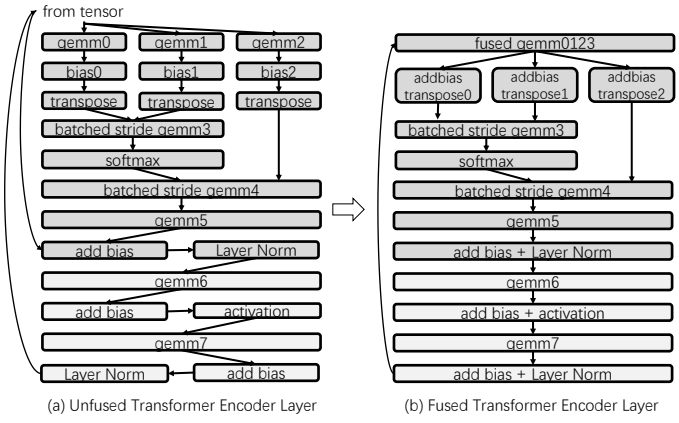

Figure 3. Kernel fusion of a transformer encoder. The part in darker color is a multi-head attention. The part in lighter color is a feed forward network.

Kernel fusion is able to increase computation efficiency by reducing the number of memory accesses, increasing cache locality, and reducing kernel launch overhead. Similar to many popular frameworks, such as TensorFlow, and Theano [2], our runtime represents the DNN forward propagation by constructing a *computation graph*, in which nodes are operators and edges are tensors. As is shown in Figure 3, the computation graph of a transformer can be reorganized into a more compact graph by fusing all the kernels between two GEMM kernels into a single one. The fused non-GEMM kernels are non-standard DNN operators, so it is impossible to take advantage of existing DNN operator libraries, such as cuDNN [21]. For example, there is no such API to combine matrix addition and transpose operation in a single CUDA kernel.

TurboTransformers implements all the non-GEMM kernels using CUDA, which can be categorized into two types. One type of kernel, such as fused activation functions and fused transpose operations, are composed of element-wise operations. There is no dependency between the processing of two different tensor elements, so we can process them in embarrassingly parallel. The second type of kernels, including Softmax and fused LayerNorm, are composed of reduction operations, which is notorious for being hard to parallelized. The latter is the focus of our performance improvement.

4.1.2 GPU-based Batch-Reduction. Both of Softmax and LayerNorm based kernels can be viewed as *Batch Reduction* operations. On the lower dimension of a 2D tensor, Softmax calculates summation and maximum and LayerNorm calculates the mean and variance. In other words, they both need to reduce a batch of 1D arrays in parallel. Table 2 shows the proportion of time of the two operators in the attention layer. In the table, the execution time of Softmax and LayerNorm is measured using PyTorch. Attention time is measured using our runtime after replaced Softmax and LayerNorm with PyTorch's implementations, respectively. We can observe that they are two big hotspots if not carefully optimized.

Table 2. Proportion of batch reduction operations in attention layer before and after optimizing.

(batch size, seq len)		(1, 10)	(1, 100)	(1, 500)	(20, 10)	(20, 100)	(20, 500)
Softmax/ Attention	before	26.23%	24.73%	34.41%	3.04%	29.4%	90.68%
	after	3.44%	3.18%	11.56%	2.46%	5.50%	15.46%
LayerNorm/ Attention	before	29.20%	21.72%	18.96%	10.61%	52.59%	83.38%
	after	4.96%	4.40%	4.08%	5.14%	6.44%	4.24%

Realizing the inefficiency of PyTorch operators, some effort is spent on optimize GPU batch reduction. According to the programming model of CUDA, efficient reduction algorithms need to fully utilize the power of three hardware levels. First, by splitting on the batch dimension, streaming-processors (SMs) level parallelism is exploited by the process of the workload on different SMs in parallel. Second, warp level parallelism is exploited by taking advantage of the

warp-level inter-thread communication mechanism provided by CUDA beyond the 9.0 version. Third, instruction-level parallelism is exploited by overlapping memory access and computation instructions. A classical implementation adopted in Faster Transformers [20] is shown in the top part of the Figure 4. It is derived from work [16], which proposed a best practice for 1-D array reduction operations on the GPU. Reduction workloads of n rows (inside the dotted line on the top of the figure) are assigned to a thread block, which is scheduled to be executed on an SM. The thread block sequentially performs n times independent 1-D array reduction, which is finished with two-pass. In the first pass, each warp of the SM conducts a warpReduce operation on 32 aligned elements, and then store reduction results of them inside shared memory. In the second pass, a warp load at most 32 partial results to register and conduct another warpReduce to obtain the final results.

In this paper, we push the efficiency boundary of classical batch reduction algorithms on GPU. Note that some space is still left for improvement of warp level and instruction parallelism in the above algorithm. First, due to the accesses of shared memory, synchronizations of warps inside an SM introduce huge overheads. Second, if the input array is not 32-aligned, warp divergence resulting from boundary processing also introduces extra overhead. Third, warpReduce leads to poor efficiency of instruction issuing. As pointed out by reference [16], there is a dependency between shuffle and add instructions. In the upper right corner of the Figure 4, the target register R3 in an SHFL DOWN instruction is required immediately as a source register in FADD instruction. The FADD instruction can only be issued until the SHFL is completely finished, whose latency is more than 1 cycle.

The above three shortcomings can be overcome by leveraging parallelism between multiple 1-D reduction operations. As shown in the bottom part of Figure 4, a new subroutine warpAllReduceSum_XElem (X = 2 in our figure) is introduced, which combine X warp as a batch and do their reduction together. First, only one synchronization is required for X elements reduction in blockReduceSum_XElem, therefore, reduces $(X-1)/X$ synchronization cost. Second, X independent boundary processing can be merged into a single one, therefore reduce warp divergence. Third, the instruction issuing is more efficient because we eliminate instructions dependency. As shown in the figure, the target register of SHFL DOWN is required two cycles later by FADD as a source register. Another SHFL DOWN with no dependency on the previous one can be issued immediately.

Especially for the LayerNorm kernel, TurboTransformers further derives a mathematical optimization trick to improve efficiency. LayerNorm requires the variances of elements of 1-D arrays. There are two equivalent formulas of variance, as shown in the Equation 1. The first one used in [20] requires two separate reductions for x and $x - E(x)$, during which one synchronization between. We use the second one

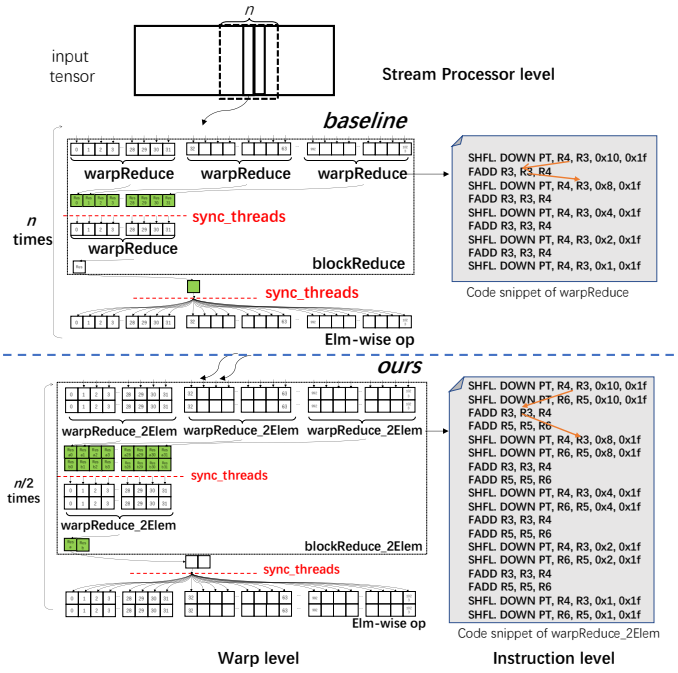

Figure 4. Parallel batch reduction optimizations on three hardware levels.

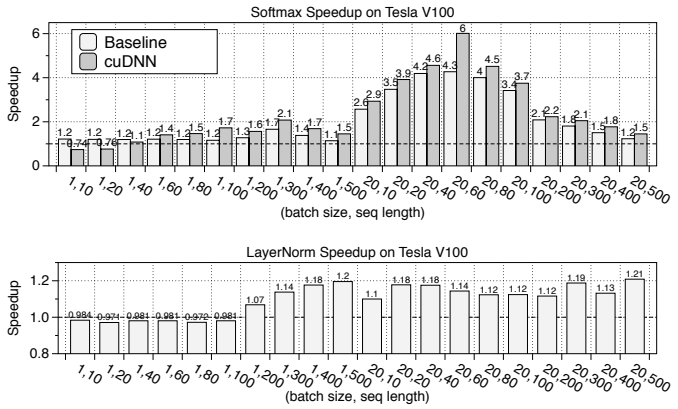

Figure 5. Speedup of batch reduction kernels on Tesla V100.

to compute variances. The warpAllReduceSum_2Elem can simultaneously reduce x and x^2, which not only increases the efficiency of instruction execution but also reduces half of the number of synchronizations.

$$\mathrm{Var}(x) = E(x - E(x))^2 = E(x^2) - E^2(x) \tag{1}$$

Figure 5 shows the speedups of Softmax and LayerNorm kernels in TurboTransformers compared with the other implementations. The baseline of both kernels is used implementations from [20]. For the Softmax kernel, we also compare

with the Softmax routine cuDNNv7.5. In most cases, our optimization strategy has achieved obvious acceleration. Due to the batch dimension is larger for Softmax, its performance boost is more significant.

4.2 Memory Manager

Besides computing optimizations, memory management is also vital to a DNN runtime. It is evaluated by two metrics. The *allocation efficiency*, which determined by the number of times and the amount of the memory is allocated and released, affects the overall execution speed of the runtime. The *memory footprint* affects the possible size of the model as well as the maximum batch size of requests. Three types of memory are managed by the runtime, i.e., input tensors, intermediate tensors, layer parameters. Dimensions of intermediate tensors change frequently during inferences in case of variable-length input. Adversely, for fixed-length input, the dimensions of intermediate tensors are determined in advance and allocation scheme never changes during inference processes. Therefore, the memory usage and positions of tensors can be pre-optimized and fixed during serving. However, The optimal memory allocation strategy is different in case of the different input lengths. Allocator of variable-length input must efficiently deal with unpredictable memory usage requirements.

It is complicated to achieve both a high allocation efficiency and a small memory footprint for variable-length input. To achieve the best memory footprint, the memory of intermediate tensors should be allocated when needed, and released immediately when not required. Frequent allocation and release of small memory space on GPU will lead to a worse runtime efficiency. For example, in this case, 50% of the computing resources idle wait for memory allocation ready on Tesla M40 (batch size = 20, sequence length = 128). To achieve the best allocation efficiency, the memory should be allocated in advance and cached for repeated use in the future the inferences without any extra allocation. However, it is challenging to predict maximum memory usage to meet the requirement of a long-term serving process. Even if we know it, occupying a vast amount of memory for a long time will lead to extremely poor memory footprints.

Although the allocator for fixed-length input has been well studied, the one for variable-length input is still not perfect. For fixed-length input, by taking advantage of the topology of the computation graph, works [24] [15] figure out the optimal memory usage by reusing the same memory space for intermediate tensors that do not coexist. For variable-length input, The existing solutions have to tradeoff allocation efficiency and footprint. PyTorch [23] designed a custom caching tensor allocator which incrementally builds up a cache of CUDA memory and reassigns it to later allocations. PaddlePaddle [17] used a similar method, which is all inspired by the caching device allocator implemented in the NVlab's cub library [26]. Experiments have shown that these methods cannot achieve optimal memory usage, because they do not consider the DNN's computation graph.

We adopted an innovative memory optimization method for variable-dimension intermediate tensors, which evoke a light-weight variable-length-aware memory manager after knowing the length of each inference. To achieve more efficiency and less footprint, the allocator used in TurboTransformers combines the idea of memory cache and graph-topology-aware space reuse. First, it organizes memory space in units of the chunks, which is a small block, for example, 2MB of memory. By reusing already allocated chunks, allocation efficiency can remain at a high level during serving. Second, it utilizes the computation graph to know the life cycle of each intermediate tensor in advance, and calculate the offset of each tensor within a specific chunk as soon as it recognizes the sequence length of the new arrival request. In this way, tensors with no overlapping life cycle can reuse the same memory space, therefore reduce memory footprint as much as possible.

Our sequence-length-aware allocation method is shown as Algorithm 1. The input tensor usage record is a list of tuples $\{first\_op, last\_op, size\}$, where $first\_op, last\_op$ are indices of the first and last operator that use the tensor. The indices are from the topological sorting of the DNN's computation graph. It first sorts the usage records in non-increasing order based on the size of the tensor. $FindGapFromchunk$ is used to determine if there exists a free gap inside a chunk to fit for that tensor. If no such gap is found in all existing chunks, we append a new chunk to the end of the chunk list. The size of the new chunk is the maximal one of DEFAULT_chunk_SIZE (2MB in our implementation) and the size of tensor times K_SCALE (1.2 in our implementation). When a chunk is not used in this inference, in our algorithm, its memory is released immediately. Alternatively, we can assign each chunk a maximum inference idle times, and release it after it reaches the time limit.

$FindGapFromchunk$ of Algorithm 1 finds the best gap in a memory chunk. It is equivalent to a special case of the 2D strip packing problem, which is NP-hard. We slightly modify the Greedy by Size for Offset Calculation Algorithm in [24] to solve FindGapFromchunk in $O(n^2)$ time complexity. The inputs are a target tensor t and a target chunk $chunk$. For each record x in the chunk, the algorithm first check whether x and the target tensor t overlap in usage time (L5-L7), in order to find the smallest gap between them such that current tensor fits into that gap (L8 - L11). If we found such a gap before the end of the chunk, we assign the tensor t to the gap. Otherwise, the function return invalid. (L15 - L21).

Figure 6 shows an example of applying our algorithm on a BERT inference application. When the input length changes from 200 to 240, we allocate one more chunk and adjust the offsets.

Algorithm 1: Sequence-length-aware allocator

def *FindGapFromchunk(t : tensor_id, chunk)*:
  get *chunk_size* from *chunk*; $smallest\_gap \leftarrow \infty$;
  $prev\_offset \leftarrow 0$; $best\_offset \leftarrow NIL$;
  foreach *record x ∈ chunk* **do**
    $max\_first\_op \leftarrow \max(first\_op_t, first\_op_x)$;
    $min\_last\_op \leftarrow \min(last\_op_t, last\_op_x)$;
    if $max\_first\_op \leq min\_last\_op$ **then**
      $gap \leftarrow offset_x - prev\_offset$;
      if $gap \geq size_t$ *and* $gap < smallest\_gap$ **then**
        $smallest\_gap \leftarrow gap$;
        $best\_offset \leftarrow prev\_offset$;
      end
      $prev\_offset \leftarrow \max(prev\_offset, offset_x + size_x)$;
    end
  end
  if *best_offset is NIL and* $chunk\_size - prev\_offset \geq size_t$ **then**
    $best\_offset \leftarrow prev\_offset$;
  end
  if *best_offset is NIL* **then**
    return *INVALID*
  end
  return *best_offset*
end
def *MemAllocate(tensor_usage_records : a list of tuples (first_op, last_op, size), chunk_list : a chunk has size, mem addr, list of <tensor_id, offset>)*:
  sort *tensor_usage_records* in decreasing order of *size*;
  foreach *record t ∈ tensor_usage_records* **do**
    $is\_assigned \leftarrow false$;
    foreach *chunk ∈ chunk_list* **do**
      $offset \leftarrow$ **FindGapFromchunk**$(t, chunk)$;
      if *offset is valid* **then**
        $assigned\_chunk_t \leftarrow chunk\_id$;
        $assigned\_offset_t \leftarrow offset$;
        $is\_assigned \leftarrow True$;
        break;
      end
    end
    if *is_assigned is false* **then**
      new_chunk_size ← max(DEFAULT_CHUNK_SIZE, *t_size*× K_SCALE);
      $assigned\_chunk_t \leftarrow$ len(*chunk_list*);
      $assigned\_offset_t \leftarrow 0$;
      append a new chunk of size *new_chunk_size* to *chunk_list*;
    end
  end
  release unused chunk in *chunk_list*;
  return *assigned_chunk, assigned_offset, chunk_list*;

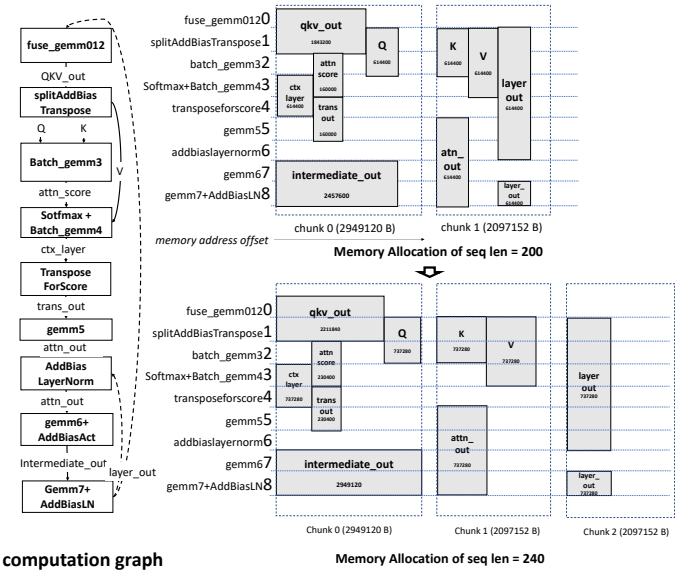

Figure 6. A memory allocation example uses our proposed variable-length-aware allocator.

5 Serving Framework

Based on the computation runtime, a serving framework is required to attain enough serving throughput under latency constraints to satisfy a Service Level Objectives (SLOs). The serving framework of TurboTransformers is shown in Figure 2. The user's requests first arrive at a *Message Queue* (MQ) and then are sent to runtime for inference computation after two serving-level optimizations, i.e. *Caching* and *Batching*. For caching, similar to Clipper [5], by caching the inference results in a database, the *Resp Cache* component in the figure responses the frequent requests without evaluating the model. For batching, the *Batch Scheduler* component is responsible for packaging requests that come in a period of time into a batch. In most application scenarios, the input of the user is a single inference request with batch size as 1. It has been known that small batch sizes lead to low GPU hardware utilization. Packaging multiple requests into a relatively larger batch and conducting inference on them together can improve hardware utilization. As shown in Figure 7, serving requests in batch brings significant speedup, especially for short sequences. The batch schedulers adopted by conventional serving systems, like TF-serving [9] and Nexus [28], are designed to packages requests with fixed-length into a batch. Currently, serving systems are lack of critical ability to handle variable-length requests.

How to batch variable-length requests to achieve the optimal throughput is tricky. If we package multiple requests of different lengths into a batch, then all requests in the batch will be zero-padded with regards to the maximum length of the request in the batch. Zero paddings will introduce a lot of extra computations. An efficient batch scheduler has to

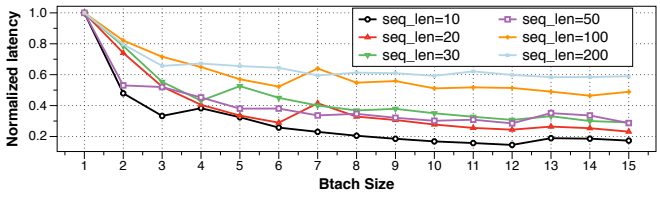

Figure 7. Batching brings performance gain for the base BERT serving on RTX 2060 GPU. The y-axis illustrates the normalized latency of inferring a request of batch size 1.

carefully balance the overheads of padding and the benefits of batching. For example, assume that there are five inference requests to be served, with lengths of 17, 18, 52, 63, and 77, respectively. Packing a single batch with 5 instances is less efficient than no batching. The batching scheme achieving the optimal throughput is packing three batches. As shown in Figure 8, the response throughput (resp/sec) improved 35% by the optimal scheduling scheme.

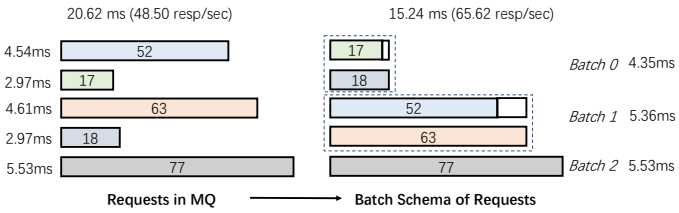

Figure 8. An example of batch scheduler for variable-length requests.

For variable-length request serving, we proposed an innovative sequence-length-aware batch scheduler in Algorithm 2. The core idea is to use dynamic programming (DP) to solve an optimization problem which maximizes the response throughput as the objective in $O(n^2)$ time complexity. The inputs of the algorithm consist of two data structures. The *request_list* is the list of input requests with variable length. The *cached_cost* is a dictionary. It uses two keys as indices, namely the sequence length and batch size. Its value is the inference cost with the parameter from the corresponding key. The values of *cached_cost* are collected by a warm-up phase after the service first starts on specific hardware, which utilizes the runtime to run inferences under all possible batch sizes and sequence lengths. They are stored on disk or database (database in Figure 2) and reloaded to memory when the serving module is restarted. First, the *request_list* are sorted in increasing order according to the sequence length. Then line 3 initializes an array called *states* to store intermediate information. Specifically, $state[i]$ records the minimum time overhead for processing the sublist $request\_list[0:i]$. The algorithm traverses each request in *request_list* and uses a DP algorithm to update the $state[i]$ at the corresponding position i. The Bellman equation of this DP problem is shown in Equation 2.

$$state[i] = \min_{0 \le j \le i} (cached\_cost[request\_list[i].len][i-j+1] \times (i-j+1) + states[j-1]) \quad (2)$$

Algorithm 2: Batch Scheduler With DP

```
Input: request_list, cached_cost
1  sort request_list in increasing order with regards to
   sequence length;
2  N ← Size(request_list);
3  Create states, start_idx_lis as lists of size N + 1;
4  states[0] ← 0; i ← 1;
5  while i ≤ N do
6      j ← i − 1; start_idx = i − 1;
7      cur_length = request_list[i − 1].length;
8      min_cost = cached_cost[cur_length][1] + states[j];
9      while j > 0 do
10         tmp_cost = states[j − 1] +
             cached_cost[cur_length][i − j + 1] ∗ (i − j + 1);
11         if tmp_cost < min_cost then
12             min_cost = tmp_cost; start_idx = j − 1;
13         end
14         j ← j − 1;
15     end
16     states[i] = min_cost; start_idx_list[i] = start_idx;
17     i ← i + 1;
18 end
19 i = N;
20 while i > 0 do
21     end_idx ← i; start_idx ← start_idx_list[i];
22     pack request_list[start_idx : end_idx] into a batch;
23     i = start_idx − 1;
24 end
```

Note that the premise of this algorithm work is that there exists a scheduling strategy for requests in MQ that can meet SLO on this server. In a multi-server environment, an upper-level load balancer as the one in Nexus [28] can ensure that the requests assigned to each server will not be overloaded. There are two options to decide when to evoke the batch scheduler. The first one is a hungry strategy. When the runtime is idle, we immediately start the batch scheduler to batch requests in MQ. This strategy is suitable for the situation where request throughput is high and the GPU have to run at full load. The second one is a lazy strategy. Similar to delayed batching of Clipper, we sets a timeout value and a maximum batch size. Once the number of requests in the batch exceeds the maximum batch size, or the timeout is reached, we start the batch scheduler. Due to the reordering of requests in Algorithm 2 , requests that arrive early may be

served late. We check the elapse between the current time and the recorded arrival timestamp of request at the front of the MQ, and start the batch scheduler immediately if the elapse plus the estimated execution latency of current requests in batch exceeds half of the latency constraints.

6 Experimental Results

We evaluated the performance of both the runtime and the serving system on a server equipped with an AMD Ryzen 7 3700X CPU and one RTX 2060 GPU.

6.1 Usability

Our runtime provides C++ and Python APIs. For the Python one, as shown in the following code snippet, adding 3 lines of python code (L3, 12, 13) can bring end-to-end speedup.

```
1  import torch
2  import transformers
3  import turbo_transformers
4  model_id = "bert-base-uncased"
5  torch_model =
       transformers.BertModel.from_pretrained(model_id)
6  torch_model.eval()
7  input_ids = torch.tensor(
8          [12166, 10699, 16752, 4454],
9          dtype=torch.long, device = torch.device('cuda:0'))
10 torch_model.to(torch.device('cuda:0'))
11 torch_res = torch_model(input_ids)
12 turbo_model =
       turbo_transformers.BertModel.from_torch(torch_model)
13 turbo_res = turbo_model(input_ids)
```

6.2 Performance of the Runtime

The runtime is evaluated on four transformer DNNs, including Bert [7], Albert [14], DistilBert [27] and a Seq2Seq Decoder [30], the parameters of which are shown in Table 3. The former three DNNs consist of Transformer encoder structures, and the latter consists of both encoder and decoder structure and is applied in a Neural Machine Translation system. The Bert model adopts a base configuration, while the Albert model adopts a large configuration.

Table 3. Evaluated transformer models & parameters

Model	Parameters
Bert	num_layer=12, num_head=12, hidden_size=4096, inter_size=3072
Albert	num_layer=12, num_head=64, hidden_size=4096, inter_size=16384
DistilBert	num_layer=6, num_head=12, hidden_size=4096, inter_size=3072
Seq2Seq Decoder	num_layer=6, num_head=16, hidden_size=3072, beam_size=4, max_target_len=500

6.2.1 End-to-end Speed on Variable-length Input. The ability to handle variable-length input is evaluated by sequential execution of the runtime using requests of different lengths. For Bert and Albert, the input requests are randomly generated texts whose lengths are uniformly distributed from 5 to 500. For Decoder, it is a model adopted from a Chinese-English translation task, and its inputs are randomly sampled Chinese texts whose length ranges from 28 to 137. The performance results are shown in Figure 9. Note that our generating and sampling processes are completely random, and the random seed is the same for different tests, although the figure displays in order of input length from small to large for the sake of clearness. We compare our runtime with the PyTorch (v1.5.0) and onnxruntime-gpu (v1.3.0). Turbo is the performance of our proposed runtime implemented by FP32 GEMM algorithms. The turbo-TC allows GEMM operation to use Tensor Core [18] if possible. Because the tensor core optimization is not allowed in PyTorch and onnxruntime, we list it here as an additional reference. However, it introduces minimal and acceptable precision loss to the FP32 version.

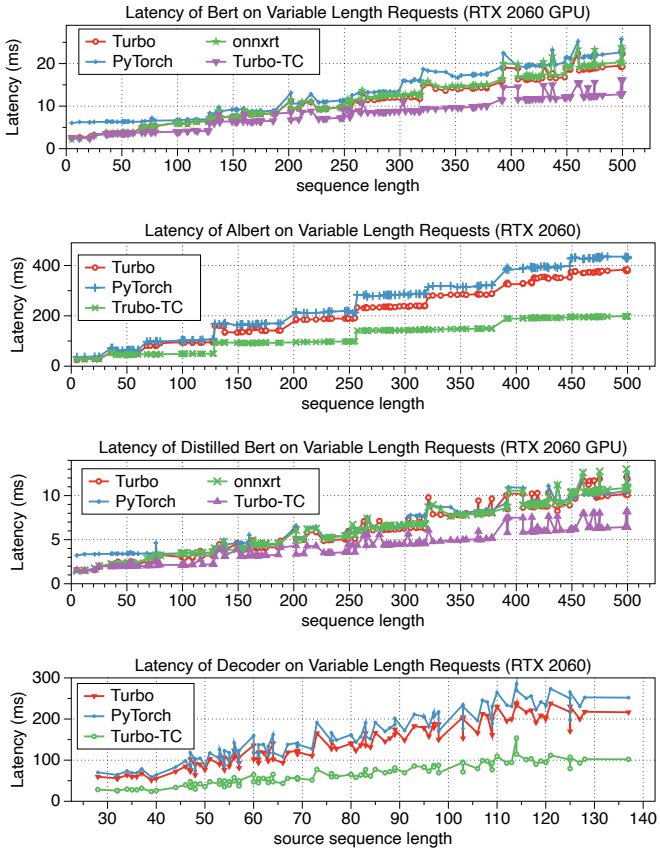

Figure 9. Benchmarking the latency of runtimes using variable length requests.

TurboTransformers' runtime shows significant performance advantages over PyTorch on short requests. In the case of Bert inference, Turbo's speedups to PyTorch are ranging from 0.97x-2.44x, on average 1.25x. In Albert inference, Turbo's speedups to PyTorch are ranging from 1.04x-1.45x, on average 1.17x. In the case of DistilBert inference, Turbo's

speedups to PyTorch are ranging from 0.85x-2.24x, on average 1.13x. In the case of Decoder inference, Turbo's speedups to PyTorch are ranging from 1.14x-1.20x, on average 1.16x. The performance improvement is more obvious in test cases of short input sequences. The latency of long input sequences is mainly limited by GEMM operations, which are not optimized by our runtime. Since the GEMM operations are conducted on the tensor core, Turbo-TC always leads to much low latency. For the same reason, the decrease in Albert's speedup to Bert is caused by the larger parameters of the Albert model, leading to an increase in the proportion of GEMM operations.

TurboTransformers exhibits similar performance with onnxruntime. In the case of Bert, the speedups of Turbo to onnxruntime are ranging from 0.88x-1.05x, on average 1.01x. In the case of DistilBert, the speedups of Turbo to onnxruntime are ranging from 0.83x-1.36x, on average 1.03x.

The time distribution of different BERT computation kernels is analyzed in Figure 10. We selected two test cases to show the inference hotspots in a long sequence input as 400 and a short sequence input as 20. In the sequence length 20, the GEMM kernels account for 70.31% of overall computation time. The Softmax kernel (ApplyMaskAndSoftmax) only accounts for 1.85% of time, and LayerNorm kernels (AddBiasLayerNorm + BertOutput/AddBiasLayerNorm) account for 2.71% of time. In the sequence length 400, the GEMM kernels account for 82.80% of overall computation time. The Softmax accounts for 4.57%, and LayerNorm accounts for 3.64%. The remaining time is spent in element-wise operations, such as activations, biases addition, tensor transposing, and reshaping. Our optimization for batch reduction is very successful since they are no longer the main hotspots among the non-GEMM kernels.

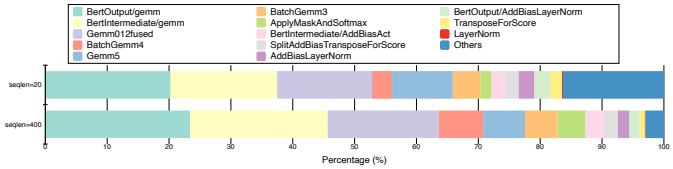

Figure 10. Time distribution of different BERT computation kernels in the case of long and short sequences.

6.2.2 Memory Optimization on Variable-length Requests.

For memory optimization, we first analyze the memory footprint of the intermediate tensors. Our proposed model-aware-allocator is compared with three other allocators, including the allocators of PyTorch and onnxruntime and an allocator implemented by Greedy by Size for Offset Calculation (GSOC) algorithm proposed in work [24], which has achieved the near-optimal memory footprint for fixed-length input inference. Memory footprint results are reported in Figure 11 and the memory allocation results are reported in Figure 12, which are evaluated using C++ APIs. The memory allocated for intermediate tensors in PyTorch and onnxruntime keep increasing during benchmarking. After processing a long sequence request, 460 in this case, the memory usage reaches its peak and no longer drops. The two runtimes incrementally build up a cache of CUDA memory and reassign it to later allocations. When there are no available fragments in the cached memory blocks, they allocate a large block of additional memory and never release it until the memory usage reaches an upper limit. In contrast, our allocator and GCOS use the information of computation-graph to control memory usage wisely. As shown in the figure, the maximum memory usage of us is 12.15 MB. The memory footprint of Turbo is quite close to the GCOS method. However, Turbo allocates and frees less memory than GCOS for each inference.

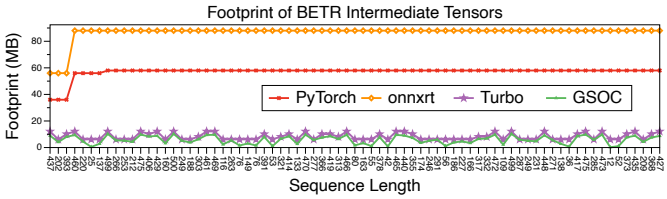

Figure 11. Footprint of intermediate tensors during BERT inferences.

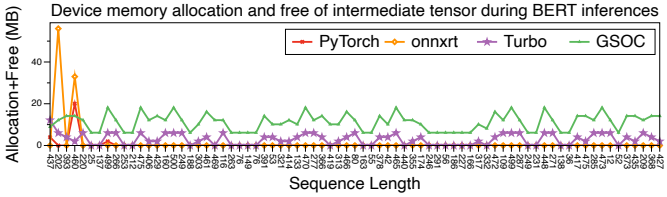

Figure 12. Amount of device memory allocation/free for intermediate tensors during BERT inferences.

We also analyze the overall memory footprint of the runtime. The peak device memory usage is measured by monitoring the nvidia-smi information in every 1 ms. the peak GPU memory used for Turbo is 663 MB while the GPU peak memory used by PyTorch is 1307 MB and 1653 MB for onnxruntime. Since the CUDA contexts of a variety of CUDA kernels need to take up a large amount of memory space [2], PyTorch and onnxruntime require more GPU memory than the DNN model and intermediate tensors actually used, which valid our efforts to implement a lightweight transformer-target inference runtime instead of a training framework or a general inference runtime.

To prove our proposed model-aware allocator's efficiency, we measure the overhead of Algorithm 1 to schedule memory offset of each intermediate tensor. Since the time complexity

[2]https://github.com/pytorch/pytorch/issues/20532

of the algorithm is O($\#tensor^2$), a trick to reduce the number of tensors is that, for the DNN model with repeated structures, we only compute memory addresses once for intermediate tensors in one of those structures and reuse them in the other ones. We measure the cost of Algorithm 1 in each inference process using a BERT model with input sequence lengths randomly generated from 5 to 500. As shown in Fig. 13., the average cost of offset scheduling (cost of Algorithm 1) is 1.8% on average (0.07%-5.77%) of DNN inference latency. The overhead of the algorithm is extremely low and its benefit outweighs its cost.

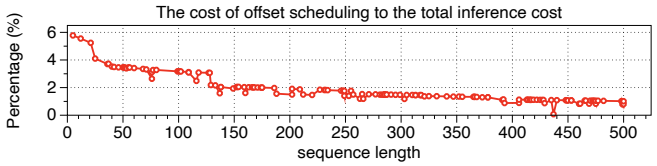

Figure 13. The offset scheduling overhead of model-aware allocator.

6.2.3 End-to-End Speed on Fixed-length Requests. We also compared with another three popular runtimes that only support fixed-length requests. 1) TensorFlow-XLA is implemented with TensorFlow (version 1.13) and preprocessed with XLA. 2) Faster Transformers (v1) is a transformer boost software developed by NVIDIA, which implements a set of customized fused kernels like us. However, it has no memory management and using the memory allocator of the TensorFlow framework. 3) NVIDIA TensorRT (v5.1.5) is an SDK for high-performance deep learning inference. The above three solutions require a time-consuming pre-tuning process based on the input dimension in advance, so they cannot be applied to handling real-time variable-length requests.

For the sake of fairness, we chose BERT as the transformer model to be evaluated on a fixed-length input task, for which every runtime has been specifically optimized for it, and an official example is provided. We select a parameter space consisting of the Cartesian Product of a set of batch sizes including 1 and 20 and a set of sequence lengths sampled from 10 to 500. The speedups of TurboTransformers to the other runtimes are shown in Figure 14. Compared with PyTorch, the speedup of Turbo is 1.23x-2.77x, on average 1.54x. Compared with onnruntime-gpu, the speedup of Turbo is 1.01x-1.26x, on average 1.11x. Compared with TensorFlow-XLA, the speedup of Turbo is 1.03x-1.31x, on average 1.11x. Compared with Faster Transformers, the speedup of Turbo is 0.71x-1.32x, on average 0.91x. Compared with TensorRT, the speedup of Turbo is 0.53x-0.96x, on average 0.87x. On average, our runtime is around 10% faster than XLA and onnxruntime and around 10% slower than Faster Transformers and TensorRT. TensorRT needs an offline tuning process, during which it can select the optimal parameters for GEMM kernels and may identify the optimal CUDA thread block sizes for non-GEMM kernels. On the contrary, our runtime does not involve the benefits of these optimizations.

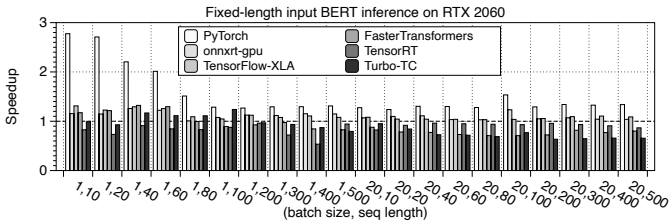

Figure 14. Benchmarking the runtimes for fixed-length input tasks on RTX 2060. The y-axis indicates normalized speedup of TurboTransformers.

6.3 Performance of Serving Framework

We choose a BERT-based service as our target application, which is used to classify a paragraph of text. The input text is first randomly generated with sequence length uniformly distributed between 2-100 and then generated with sequence length uniformly distributed between 5-500. The requests are sent to the serving system with *Poisson* inter-arrival times. We turn off the caching optimization.

There are two strategies to build the cached_cost dictionary in Algorithm 2. First, if the parameter space is small, a warmup phase that records the latency of executing with all possible parameters is required after the service started. It takes tens of minutes in our experiments. Second, if the parameter space is large, we sample the parameter space and use the interpolation method to estimate a specific case during serving. After you get real data, it can be used to update the dictionary in a lazy evaluation way.

The serving throughput of input sequence length ranging 2-100 is illustrated in Figure 15. The figure's x-axis indicates how many requests arrive at the serving system per second, ranging from 20 req/sec to 1500 req/sec. The figure's y-axis represents how many responses can be obtained per second, which is usually called serving throughput. The *Critical Point* for service latency to remain stable is that request throughput, and serving throughput are equal. When request throughput is higher than the critical point, the requests will accumulate in the message queue, leading to long delays of latter requests. After a while, its latency will gradually tend to infinity ($+\infty$), and the service system has to drop some requests.

Our proposed batch scheduler achieves the best serving throughput. The baseline is shown as PyTorch-Nobatch, which uses PyTorch as the runtime and serve without batching optimization. Turbo-NoBatch is the service using our proposed runtime to replace PyTorch. Turbo-Naive-Batch is implemented with a naive batch scheduler, which packs the requests currently inside the message queue into a single batch.

Turbo-DP-Batch is our proposed variable-length-aware batch scheduler. The batch scheduler of our system employs the hungry strategy, and the maximum batch size is 20. The serving throughput of PyTorch-Nobatch is saturated at 99 resp/sec. The serving throughput of Turbo-NoBatch is saturated at 237 resp/sec (**2.39x**). Turbo-Naive-Batch improves it to 323 resp/sec (**3.26x**) and the Turbo-DP-Batch futher improves it to 402 resp/sec (**4.06x**).

The latency results on four systems' critical points are shown in Table 4. Since batching brings higher GPU utilization for short requests, the latency of Naive-Batch is smaller than NoBatch. Turbo-DP-Batch sorts the requests in MQ, the execution of long sequences is delayed, thus increasing their latency. Therefore Naive-Batch brings smaller latency than Turbo-DP-Batch.

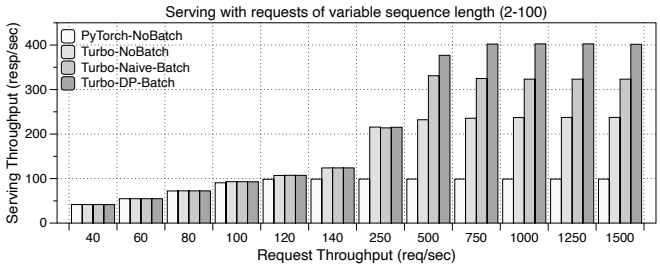

Figure 15. Serving throughput under different request throughput (Sequence Length 2-100).

Table 4. The latency of four serving systems (sequence length 2-100). Table item contains the avg (min, max) latency in ms.

Request Thrpt (req/sec)	PyTorch NoBatch	Turbo NoBatch	Turbo Naive-Batch	Turbo-DP-Batch
99	36.52 (10.05, 108.72)	7.47 (3.84, **18.59**)	7.51 (**3.78**, 21.58)	**7.49** (3.82, 20.23)
237	+∞	16.68 (3.68, 45.39)	**10.09** (**3.67, 28.60**)	10.89 (3.81, 37.00)
323	+∞	+∞	**12.44** (**3.48, 26.70**)	15.37 (3.58, 40.45)
402	+∞	+∞	+∞	**24.74** (**4.23, 57.02**)

When increasing the length difference between the sequences by changing the input sequence range from 2-100 to 5-500, our proposed allocator achieves the best throughput and leads to the lowest latency. The throughput of input sequence length ranging 5-500 is illustrated in Figure 16 and the latency results are shown in Table 5. In this case, we turn on the tensor core optimization, which brings no accuracy loss and can better satisfy the SLO. The serving throughput of PyTorch-Nobatch is saturated at 60 resp/sec, while Turbo-TC-NoBatch improves it to 120 resp/sec (**2.0x**). Turbo-TC-DP-Batch further improves it to 144 resp/sec (**2.4x**). Due to the additional zero-padding overhead, Turbo-TC-Naive-Batch's throughput is 98 resp/sec, which is even worse than Turbo-NoBatch. As shown in Table 5. Under the same request throughput rate, Turbo-DP-Batch usually brings the lowest average and maximum service latency. This is because of the reduction of zero-padding cost and shortening the latency of every single request.

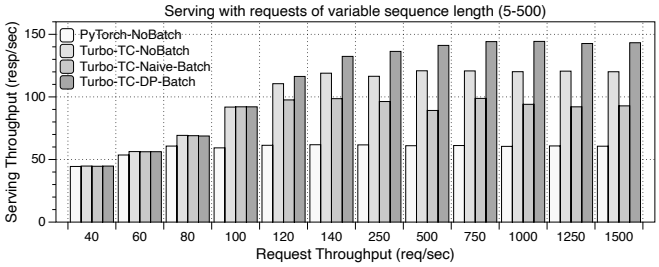

Figure 16. Serving throughput under different request throughput (Sequence Length 5-500).

Table 5. The latency of four serving systems (sequence length 5-500). Table item contains the avg (min, max) latency in ms.

Request Thrpt (req/sec)	PyTorch NoBatch	Turbo NaiveBatch	Turbo NoBatch	Turbo-DP-Batch
60	77.71 (10.61, 158.06)	17.80 (3.06, 121.96)	**8.05** (2.76, **20.53**)	**8.05** (**2.70**, 27.42)
98	+∞	16.68 (2.96, 65.09)	24.88 (3.0, 65.09)	**13.79** (**2.94, 45.09**)
120	+∞	+∞	32.91 (3.14, 127.61)	**23.18** (**2.72, 81.83**)
144	+∞	+∞	+∞	**38.51** (**4.44, 106.65**)

7 Conclusion

TurboTransformers improves latency and throughput for deploying transformer-based DL services in GPU datacenter. It solves two critical problems introduced by the transformer models, which are unprecedented computation pressure and variable-length input. For these purposes, it proposed three innovations in computing, memory and serving levels, including a new parallel batch reduction algorithm for Softmax and LayerNorm kernels, a sequence-length-aware memory allocator as well as a sequence-length-aware batch scheduler. The runtime achieves better speed than PyTorch and similar speed as onnxruntime in the variable-length request tests but maintains a smaller memory footprint. It also achieves comparable speed than TensorFlow-XLA, TensorRT, and FasterTransformers in the fixed-length request tests. While conventional batching is inefficient for variable-length request, the serving framework achieves higher throughput using the proposed batch scheduler.

8 Acknowledgements

We would like to thanks Yin Li (UC Davis), Shengqi Chen, Wentao Han (Tsinghua Univ.) for their proofreading.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI16)*. 265–283.

[2] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, Vol. 4. Austin, TX, 1–7.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[4] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259* (2014).

[5] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 613–627.

[6] Weihao Cui, Mengze Wei, Quan Chen, Xiaoxin Tang, Jingwen Leng, Li Li, and Mingyi Guo. 2019. Ebird: Elastic Batch for Improving Responsiveness and Throughput of Deep Learning Services. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE, 497–505.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[8] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[9] Google. 2020. TensorFlow Serving. https://github.com/tensorflow/serving. [Online; accessed July-2020].

[10] Google. 2020. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla. [Online; accessed July-2020].

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[12] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[14] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942* (2019).

[15] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. 2019. On-device neural net inference with mobile gpus. *arXiv preprint arXiv:1907.01989* (2019).

[16] Yuan Lin and V Grover. 2019. Using cuda warp-level primitives. *Retrived from https://devblogs. nvidia. com/using-cuda-warp-level-primitives/. Accessed* (2019).

[17] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. 2019. PaddlePaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing* 1, 1 (2019), 105–115.

[18] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 522–531.

[19] MicroSoft. 2020. ONNX Runtime is a cross-platform inferencing and training accelerator compatible with many popular ML/DNN framework. https://github.com/microsoft/onnxruntime. [Online; accessed July-2020].

[20] NVIDIA. 2019. FasterTransformer V1, a highly optimized BERT equivalent Transformer layer for inference. https://github.com/NVIDIA/DeepLearningExamples/tree/master/FasterTransformer. [Online; accessed April-2020].

[21] NVIDIA. 2020. cuDNN. https://developer.nvidia.com/cudnn. [Online; accessed Augest-2020].

[22] NVIDIA. 2020. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt. [Online; accessed July-2020].

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. 8026–8037.

[24] Yury Pisarchyk and Juhyun Lee. 2020. Efficient Memory Management for Deep Neural Net Inference. *arXiv preprint arXiv:2001.03288* (2020).

[25] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI Blog* 1, 8 (2019), 9.

[26] NVIDIA Research. 2020. CUB. https://github.com/NVlabs/cub. [Online; accessed Augest-2020].

[27] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).

[28] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 322–337.

[29] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[31] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*. 5753–5763.

A Artifact Appendix

A.1 Abstract

The artifact contains the code for the runtime of TurboTransformers. We provide instructions for obtaining critical results used in this paper and scripts for running the experiments in the paper.

A.2 Description

A.2.1 Check-list (artifact meta information).

- **Algorithms**: The artifact includes the runtime component of TurboTransformers. More specifically, it contains the parallel batch-reduction algorithms proposed in Section 4.1.2, as well as the model-aware-allocator proposed in Section 4.2.

- **Datasets**: The inputs used for benchmark scripts of the artifact are randomly generated.
- **Compilation**: The experiments in the paper used g++ version 7.5.0, nvcc version 10.2. The artifact also provided a docker file to build a docker environment to compile the code. The docker version used is 19.03.8.
- **Runtime environment**: The artifact provided a compilation script to run on the CentOS 7 OS installed with CUDA version 10.2.
- **Hardware**: The artifact can run on a server equipped with at least one NVIDIA GPU. The supported GPU microarchitecture codenames include Pascal, Volta, Turing.
- **Output**: Running times of the algorithms are output to the console.
- **Experiment Workflow**: Clone the repository and use the provided scripts to run the experiments.
- **Publicly available**: Yes

A.2.2 How Delivered. Available as open-source under the BSD license: https://github.com/Tencent/TurboTransformers. The artifact branch is ppopp21_artifact_centos.

A.3 Installation

A.3.1 Docker.

- Build a docker image using the docker file.

```
bash tools/build_docker_gpu.sh $PWD
```

- Run the image as a container

```
nvidia-docker run --gpus all
    --net=host --rm -it -v
    $PWD:/workspace
    --name=turbo_dev:latest
```

- Inside the container, build the artifact.

```
cd /workspace
bash
    tools/build_and_run_unittests.sh
    $PWD -DWITH_GPU=ON
```

A.3.2 CentOS.

- Use the following command to build the artifact on CenOS 7.

```
bash build_centos.sh
```

A.4 Experiment Workflow

You can run the scripts provided in the benchmark directory to compare the speed of turbo runtime with PyTorch. Run the following script will reproduce the Bert and Albert results shown in Figure 9.

```
bash gpu_run_variable_benchmark.sh
```

Run the following script will reproduce the Bert and Albert result in Figure 14.

```
bash gpu_run_fixed_benchmark.sh
```