PetS: A Unified Framework for Parameter-Efficient Transformers Serving

Zhe Zhou, *Peking University;* Xuechao Wei, *Peking University and Alibaba Group;* Jiejing Zhang, *Alibaba Group;* Guangyu Sun, *Peking University*

https://www.usenix.org/conference/atc22/presentation/zhou-zhe

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

NetApp®

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

PetS: A Unified Framework for Parameter-Efficient Transformers Serving

Zhe Zhou [†]
Peking University

Xuechao Wei
Peking University
Alibaba Group

Jiejing Zhang
Alibaba Group

Guangyu Sun [*]
Peking University

Abstract

Deploying large-scale Transformer models under the conventional *pre-train-then-fine-tune* paradigm is impractical for multi-task serving, because a full model copy for each downstream task must be maintained, quickly exhausting the storage budget. Recent algorithmic advances in Parameter-Efficient Transformers (PETs) have shown enormous potential to mitigate the storage overhead. They share the pre-trained model among tasks and only fine-tune a small portion of task-specific parameters. Unfortunately, existing serving systems neither have flexible PET task management mechanisms nor can efficiently serve queries to different tasks in batches. Therefore, we propose PetS, the first unified framework for multi-task PETs serving. Specifically, different PET tasks are expressed by a unified representation in the same framework, which enables flexible PET task management. Based on the unified representation, we design a specialized PET inference engine to batch different tasks' queries together and execute them with task-agnostic shared operators and task-specific PET operators. To further improve system throughput, we propose a coordinated batching strategy to deal with arbitrary input queries. We also develop a PET operator scheduling strategy to exploit parallelism between PET tasks. Comprehensive experiments on Edge/Desktop/Server GPUs demonstrate that PetS supports up to 26× more concurrent tasks and improves the serving throughput by 1.53× and 1.63× on Desktop and Server GPUs, respectively.

1 Introduction

Recently, large-scale pre-trained Transformer models have revolutionized the field of artificial intelligence. Benefited from the practical *pre-train-then-fine-tune* paradigm, Transformer models such as Bert [9], GPT-2/3 [3,45], Roberta [31], XLNet [59], T5 [46], and some other variants [28,29] have achieved the leading-edge performance on various NLP (Natural-Language-Processing) tasks, including question-answering, sentiment-classification, text classification and machine translation, etc. Besides NLP tasks, some recent works also apply transformers to computer vision tasks [4,11,25,32,55,61], which demonstrate comparable or even superior performance against conventional Convolutional Neural Networks (CNNs). In brief, Transformers have been recognized as a milestone of artificial intelligence.

To date, a standard workflow has been shaped to apply Transformers to real-world applications. As is depicted in Figure 1, big companies like Google first pre-train the Transformer models like Bert [9] and GPT [3,45] with large-scale datasets (Step ①). The unsupervised pre-training usually lasts for days to months, even trained on TPU clusters [3,9]. The pre-trained models with rich task-agnostic knowledge are provided to application developers, who then fine-tune the pre-trained models on their private datasets in a supervised manner (Step ②). The fine-tuned task-specific models are finally deployed to cloud or edge servers (Step ④) to process different input queries. Such a workflow, however, is faced with the poor scalability issue in the pervasive multi-task serving scenarios [18,19,34,40,48,53]. Since application developers fine-tune and maintain a full model copy for each downstream task, the storage overhead is proportional to the number of deployed tasks. Considering the enormous parameters (e.g., several hundred millions to several thousand millions of parameters) of Transformer models, the storage overhead will be huge. What is worse, conventional serving frameworks have to swap in and out models frequently if the GPU memory cannot hold all the invoked tasks, resulting in much lower serving throughput. Also, since the input queries are associated with different models, we cannot inference them in batches for higher serving throughput [7,12,16,50].

Recent algorithmic advances in Parameter-Efficient Transformers (PETs) have shown enormous potential to solve these problems partially. They share the pre-trained model weights among tasks and only fine-tune a small portion of task-specific parameters for each downstream task [14,20,23,24,41,62,64]. By this means, the storage overhead is substantially mitigated, while the model accuracy is still comparable or even superior to the full-model fine-tuning counterparts. These methods, however, cannot run efficiently with existing Transformers serving frameworks [12,37,56]. On the one hand, due to the lack of PET task management mechanism and PET-oriented inference engine, we have to merge PET parameters into the shared model and still send full model copies to the frame-

[†] Work done during Zhe Zhou's internship at Alibaba DAMO Acadamy.
[*] Corresponding author.

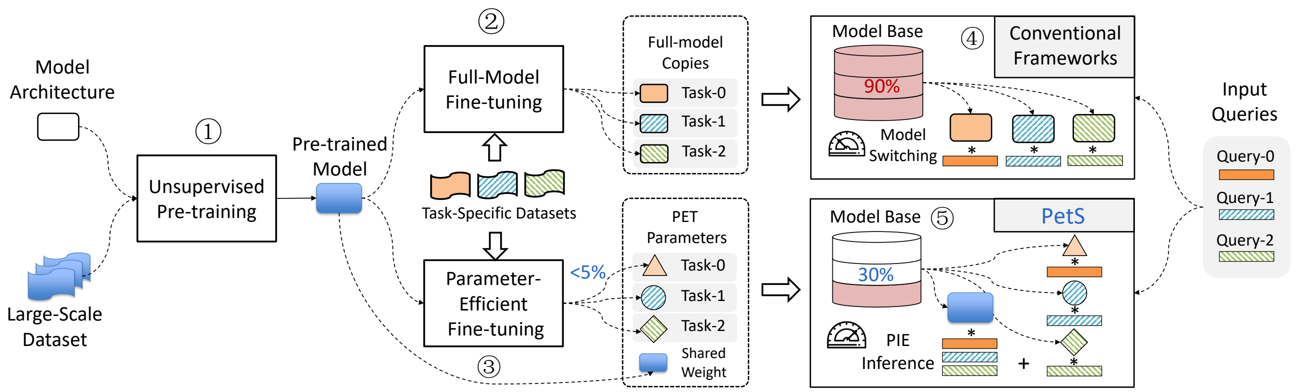


Figure 1: The conventional workflow VS. PetS workflow for developing and deploying Transformer-based applications. ①: Large-scale unsupervised pre-training. ②: Full-model fine-tuning on customized datasets for different tasks. ③: Parameter-efficient fine-tuning. ④: Serving the input queries with conventional frameworks.⑤: Serving the input queries with PetS.

works. Thus, the GPU memory footprint is not mitigated. On the other hand, queries to different tasks cannot be processed in batches due to both the inter-task weight differences and inter-algorithm representation differences.

To take full advantage of parameter-efficient Transformers, in this paper, we propose PetS, a unified framework for multi-task PETs serving with extraordinary scalability and performance. To this end, we first express the state-of-the-art PET algorithms by a unified representation, which decouples any PETs into task-agnostic shared operations and task-specific PET operations. Based on the unified representation, we design a PET tasks management mechanism, which enables the service providers to register and load PET tasks flexibly. We then develop a high-performance PET Inference Engine (PIE) to batch different tasks' queries and execute them with shared operators and light-weighted PET operators, substantially improving the serving throughput. We also propose several optimization strategies to improve the system's throughput further. To be specific, we propose a Coordinated Batching strategy to deal with arbitrary input queries (i.e., queries with different sequence lengths and PET types). To exploit parallelism between PET operators, we apply a PET Operator Scheduling strategy to properly put concurrent PET operators to different CUDA streams. We comprehensively evaluate PetS on Edge/Desktop/Server GPU platforms. Compared to conventional frameworks, PetS supports up to 26× more concurrent Transformer tasks and improves the serving throughput by 1.53× and 1.63× on Desktop and Server GPUs, respectively. Therefore, PetS shows great potential to reduce the service deployment cost and improve the service quality in multi-task Transformers serving scenarios.

对不同 tasks 的 queries 进行 batch, 并解耦为 shared ops 和 轻量 PET ops.

处理了不同 seq_len 和 types 的 queries

2 Background & Motivations

2.1 Transformer Models

As illustrated in Figure 2, Transformer models are generally built by stacking several homogeneous Transformer blocks. A standard Transformer block consists of three key components: Multi-Head Attention (MHA), Feed-Forward Network (FFN) and Normalization Layers (Norm). For each block, the input is a sequence of n vectors (tokens), denoted as $X \in \mathbb{R}^{n\times d_{in}}$, where n and d_{in} are the sequence length and the input feature dimension. Three linear projection weights $\{W_Q, W_K, W_V\} \in \mathbb{R}^{d_{in}\times d}$ project the input tensor X to Query, Key and Value tensors, denoted as $\{Q, K, V\} \in \mathbb{R}^{n\times d}$ (Step ① in Figure 2), where d represents the hidden feature dimension. The Q, K, V tensors are split to multiple "Heads" (Step ②) to perform softmax-based self-attention respectively (Step ③). The self-attention results are then concatenated (Step ④) and linear-transformed (Step ⑤) to generate the MHA results. After skip-connection and layer-normalization, the hidden feature X is then fed into the FFN layer, which computes with two fully-connected layers (Step ⑥ and ⑦). The GELU activation function [22] is applied to the first layer's output. One block's output serves as the input of the next block. On top of the last block, there is usually a classification layer to generate the final results for a given downstream task.

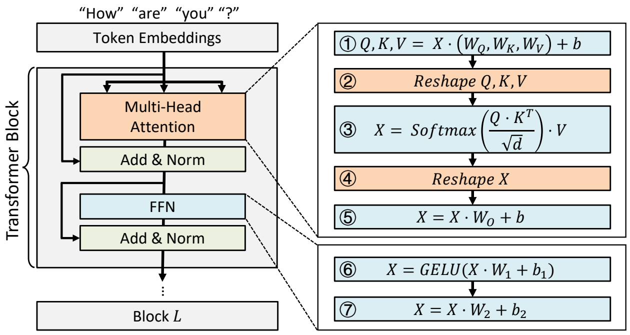
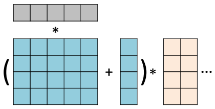
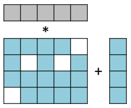
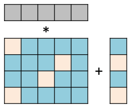
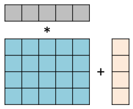


Figure 2: Bert Architecture.

Traditional neural-networks like CNNs and LSTMs all involve "prior" in their models to enhance the performance (i.e., a CNN model assumes the 2D images have spatial locality, while an LSTM model assumes that the information should be either remembered or forgotten). In comparison, Transformers have no such priors and learn all the useful information

Table 1: Comparisons of State-of-the-Art Parameter-Efficient Transformers.

Model Type	Adapters [23]	MaskBert [64]	Diff-Pruning [20]	Bitfit [62]
Main Computation. Pre-trained Parameters Are Marked Blue				
Formula	$Y_t = X_t \cdot W + b$, $Y_t = Y_t + \sigma(Y_t \cdot W_{\text{down}}) \cdot W_{\text{up}}$	$Y_t = X_t \cdot (W \odot M_t) + b$	$Y_t = X_t \cdot (W + \delta_t) + (b + b_t)$	$Y_t = X_t \cdot W + b_t$
Additional Parameters	7.3%	3%	0.5%	3.8%
Leading Tasks in GLUE Benchmark *	STS-B, QQP	SST-2	QNLI, MNLI, CoLA,MRPC	RTE

*Comparisons are based on BERT-large. We reproduce MaskBert on BERT-large, while the other results are obtained from the reported numbers.

purely through unsupervised pre-training. To achieve this, not only the pre-train datasets are large in scale, the models also contain enormous parameters to guarantee a high knowledge capacity. For instance, Bert-base and Bert-large have 110M and 340M parameters, respectively, while some recent models even have billions [3, 51] or trillions of parameters [13]. The explosion of such large-scale Transformer models brings both opportunities and challenges. On the one hand, real-world tasks get benefited from their superior performance compared to traditional DNNs. On the other hand, it is challenging to deploy Transformer models to resource-constrained scenarios due to the storage/memory capacity limit, especially when multiple tasks should be served simultaneously.

2.2 Multi-Task Transformers Serving

In real-world scenarios, a server usually runs multiple tasks (here a *task* refers to a distinct DNN model) concurrently for serving different queries [2, 48, 50] (each query invokes at least one of the DNN models). According to the standard workflow shown by steps ①-②-④ in Figure 1, for multi-task transformers serving, every downstream task has its own fine-tuned model. That is to say, the storage/memory overhead is proportional to the number of tasks. In the figure, three tasks occupy 3× storage. More importantly, all the models should be buffered in GPU memory for quick response to different queries. As the number of tasks increases, it will easily exceed the GPU's memory. Alternatively, we can swap in and out models once some tasks are invoked. Such a method, however, will downgrade the system's performance due to the considerable model swapping overhead [19, 48, 53]. Also, if each task only has limited input queries, the computation resources will be under-utilized because of the small batch size.

Although previous serving systems/frameworks like INFaaS [48], Nexus [50], Rafiqi [53], Triton [40], Tensorflow Serving [18] and many DNN accelerators [2, 6, 17, 26, 27] have emphasized the multi-task DNN serving ability, to implement multi-task Transformers serving is still challenging. Reasons are mainly two-folded. First, Transformers usually contain enormous parameters to guarantee their sufficient knowledge capacity. Thus, the storage and memory overhead is much heavier than traditional DNNs, limiting the number of served tasks. Second, previous multi-task inference frameworks/accelerators assume that the computation/bandwidth requirements vary among concurrent DNNs. Thus, they execute computation-bounded and memory-bounded models (or layers) together to fully utilize the hardware resources. However, since the Transformer blocks are homogeneous among different tasks, there is little room for improving system throughput by co-locating heterogeneous models.

2.3 Parameter-Efficient Transformers

A potential solution to the multi-task Transformers serving problem is directly training a multi-task model like T5 [46]. However, such a method is infeasible in real scenarios since all the application developers have to provide their private datasets to train such a one-for-all model. Recently, Parameter-Efficient Transformers (PETs) have emerged as another promising way to deal with the problem. PETs are based on the assumption that pre-trained models have learned rich knowledge from large-scale pre-train datasets [44, 47]. Thus, we can adapt the pre-trained model to downstream tasks by only fine-tuning a small portion of task-specific parameters rather than the whole model. As illustrated in Figure 1, through parameter-efficient fine-tuning (Step ②), only the PET parameters should be stored for each downstream task. For example, four representative PETs, namely Adapters [23], MaskBert [64], Diff-Pruning [20], and Bitfit [62] only use 0.5% to 7.3% additional parameters for each task. However, they still achieve comparable or even higher accuracy against the full-model fine-tuning counterparts. We summarize them in Table 1 and introduce them as follows:

Adapters: Adapters [23] proposes to inject trainable, task-specific "adapter" modules between some layers of the pre-trained model, while the pre-trained weights are shared among tasks. Formally, assume the linear layers in a pre-trained model compute the hidden feature Y_t with input feature X_t and pre-trained parameters W (weight) and b (bias), namely $Y_t = X_t \cdot W + b$, then an adapter module manipulates the hidden features with two learnable weights $W_{down} \in \mathbb{R}^{d \times d_m}$ and $W_{up} \in \mathbb{R}^{d_m \times d}$, namely $Y_t = Y_t + \sigma(Y_t \cdot W_{down}) \cdot W_{up}$, where σ is the activation function. Since the bottleneck dimension $d_m << d$, the Adapter modules are small in size. Each task only requires about 7.3% of new parameters (including a task-specific classification layer).

只面向 Linear layers，transformer-based model 里面 line layers 的输入 (i.e., attention on V 的计算) 为向量

MaskBert: Based on the lottery ticket hypothesis on Bert [5, 43], MaskBert [64] adapts the pre-trained model to downstream tasks by learning binary masks for each weight matrix. As shown in Table 1, for each task, the pre-trained model (including the classification layer) is frozen. Only the binary masks with about 5% of zero elements are learned for each weight matrix. Since the masks are binary, MaskBert only incurs about a 3% per-task storage overhead. For each linear layer, the computation is represented as $Y_t = X_t \cdot (M_t \odot W) + b$, where M_t denotes the task-specific mask.

Diff-Pruning: Diff-Pruning [20] also shares the pre-trained model among tasks and only fine-tunes a small portion of "difference" for each downstream task. As shown in Table 1, the orange elements in both weight and bias represent the fine-tuned "difference", which only incur about 0.5% of new parameters for each task. During inference, these difference parameters, denoted as δ_t and b_t, are merged with the pre-trained model to construct a task-specific model. Thus, the main computation is $Y_t = X_t \cdot (W + \delta_t) + (b + b_t)$.

Bitfit: Besides a task-specific classification layer, Bitfit [62] only fine-tunes the linear and normalization layers' bias-terms, which also achieves competitive accuracy on some tasks in the standard GLUE benchmark [54]. As shown in Table 1, the linear layers in Bitfit compute with $Y_t = X_t \cdot W + b_t$ where b_t is the only task-specific parameter.

There are still many other emerging PET algorithms [14, 24, 33, 41]. Their workflows are similar to at least one of the above PETs. Therefore, in this paper, we conduct the discussion mainly based on these four representative PETs.

2.4 Challenges of Multi-Task PETs Serving

When serving T different tasks, PETs reduce the storage overhead from original $T \times \gamma$ to $T \times \eta + \gamma$, where γ and η denote the amount of full-model parameters and PET parameters, respectively. Since $\eta << \gamma$, using PETs can significantly reduce the storage overhead. However, we notice that the algorithmic advantages of PETs can hardly translate to real speedup with conventional Transformers serving frameworks, mainly due to the following challenges:

Challenge #1: Current frameworks cannot support various PET algorithms flexibly. We present the leading GLUE tasks of each PET algorithm in Table 1. As we can see, all PETs have their advantageous tasks. None of the four PETs can serve as the one-for-all choice. That is to say, the application developers tend to choose the best PETs for their downstream tasks [33]. Therefore, the serving framework has to support multiple types of PETs. However, current serving frameworks are not optimized for diverse PETs. They lack the mechanism to register and manage different PETs flexibly, considering their distinct algorithmic representations.

Challenge #2: The GPU-memory footprint is still not mitigated. To serve PETs like MaskBert and Diff-Pruning using conventional inference frameworks, we have to merge the task-specific PET parameters into the shared model. After that, we load the newly constructed models to inference frameworks for serving. Thus, concurrent tasks still occupy $O(T \times \gamma)$ GPU memory, limiting the system's scalability. As discussed before, we can swap in/out models among tasks to deal with the GPU capacity issue, which will result in low throughput due to the model swapping overhead [19, 48, 53].

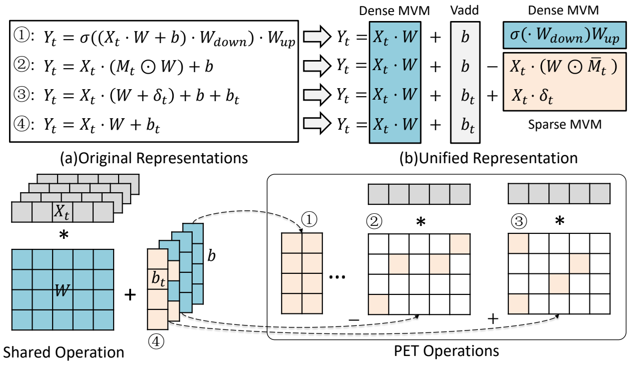

Dense MVM
Vadd
Dense MVM
①: $Y_t = \sigma((X_t \cdot W + b) \cdot W_{down}) \cdot W_{up}$
②: $Y_t = X_t \cdot (M_t \odot W) + b$
③: $Y_t = X_t \cdot (W + \delta_t) + b + b_t$
④: $Y_t = X_t \cdot W + b_t$
$Y_t = X_t \cdot W + b$ $\sigma(\cdot W_{down})W_{up}$
$Y_t = X_t \cdot W + b - X_t \cdot (W \odot \bar{M}_t)$
$Y_t = X_t \cdot W + b_t + X_t \cdot \delta_t$
$Y_t = X_t \cdot W + b_t$
Sparse MVM
(a)Original Representations
(b)Unified Representation
X_t
*
W
+
b_t
b
①
②
③
④
*
*
...
−
+
Shared Operation
PET Operations

Figure 3: Unified representation of PETs. ①: Adapters ②: MaskBert ③: Diff-pruning ④:Bitfit.

Challenge #3: It is still hard to improve the system's serving throughput, especially when each task only has limited queries. It is well-known that batched inference is a practical technique to improve a DNN serving system's throughput [7, 12, 16, 50]. However, due to the differences in PET parameters and PET algorithms, conventional frameworks can hardly batch different tasks' queries (even though the tasks may belong to the same PET algorithm) for higher throughput. Such a problem will be more prominent when each concurrent task only has a few queries to process.

3 `PetS` Framework

To address the challenges outlined above, we propose `PetS`, a unified framework for efficient multi-task PETs serving. We first propose a unified representation to put all PETs into one framework. Based on this, we develop a flexible PET tasks management mechanism and a specialized PET Inference Engine (PIE) that enables both inter-task and inter-algorithm query-batching. Details are introduced as follows.

3.1 Unified Representation of PETs

As Table 1 shows, state-of-the-art PETs have different algorithmic representations, resulting in a "fragmentation" problem. We propose a unified representation, which expresses PETs with task-agnostic and task-specific operations, to help put them into one framework and enable batched inference. As illustrated in Figure 3, for each PET, we decouple the main computation (linear layers) into three operations: (1) Dense Matrix-Vector-Multiplication (MVM) operation using shared pre-trained weights. (2) Bias vector addition (Vadd) using shared or task-specific bias. (3) Sparse/dense MVM operations using task-specific PET parameters. Since all PETs share the same pre-trained weight matrix W, the first operation, namely $X_t \cdot W$ can be batched together. Though the task-specific computation with PET parameters cannot be batched among PETs, it only involves light-weighted operations.

Adapters and Bitfit naturally fit into such a representation, since the PET operations are already decoupled from the shareable operations. For Diff-Pruning and MaskBert, we need to perform some equivalent transformations. As Figure 3 shows, for Diff-Pruning, the computation concerning the shared weight and "difference" are conducted separately. Then the results are added up, namely $X_t \cdot (W + \delta_t) = X_t \cdot W + X_t \cdot \delta_t$. For MaskBert, we use an equivalent transformation: $M_t \odot W = (1 - \overline{M}_t) \odot W$, where $\overline{M}_t$ denotes the bit-wise inversion of binary mask M_t. Thus, the original MVM operation is converted to $X_t \cdot W - X_t \cdot (W \odot \overline{M}_t)$. The $W \odot \overline{M}_t$ term can be treated as the sparse weight differences similar to Diff-Pruning's. Since δ_t and $W \odot \overline{M}_t$ are sparse matrices with high sparsity (typically 95% - 99.5%), these PET operations can be efficiently computed with sparse kernels. Considering that the Vadd operations concerning bias terms only have little overhead, we mainly focus on operations (1) and (3), namely the sparse and dense MVM operations.

The unified representation brings two main advantages. First, the queries from different tasks can be batched together at step (1), regardless of the PET types. Let us consider an extreme case: assume we have T tasks, each having a single query. The execution latency is changed from $\sum_{i=0}^{T-1} \alpha(1)$ to $\sum_{i=0}^{T-1} \beta_i + \alpha(T)$, where $\alpha(n)$ denotes the latency of running a Transformer model (without PET) with a batch of n queries. β_i denotes the latency of PET operations for query i. The throughout is improved if we have:

$$\sum_{i=0}^{T-1} \beta_i + \alpha(T) < \sum_{i=0}^{T-1} \alpha(1) \tag{1}$$

The inequality always holds since $\beta_i << \alpha(1)$ (the PET operations are light-weighted compared to the shared operations) and $\alpha(T) << \sum_{i=0}^{T-1} \alpha(1)$ (batched inference can greatly reduce the average latency [7, 12, 16, 50]).

Second, such a unified representation simplifies the PET tasks management. Each task can be registered by identifying its shared model tag, PET type, and PET parameters. The inference engine can then load these PETs in a unified way.

3.2 Framework Overview

Based on the unified representation, we then present the PetS serving framework to support the management and serving of PET tasks. Figure 4 illustrates the proposed PetS framework. PetS has three main components: a Task Manager, a Parameter Repository, and a PET Inference Pipeline. PetS works as follows: ❶: The framework first registers the PET tasks submitted by developers. For each PET task, the developers are required to provide the Pre-trained Model Tag (such as `bert-base-cased`), PET Parameters (in compressed format) and PET Type (e.g., `MaskBert`). ❷: Task Manager registers PET tasks, which assigns a unique `Task_id` to each submitted task. The PET parameters and the pre-loaded shared model parameters are all stored in the Parameter Repository (❸). After registration, the PET Inference Engine (PIE) is responsible for processing the input queries (❹) through an optimized PET Inference Pipeline (❺).

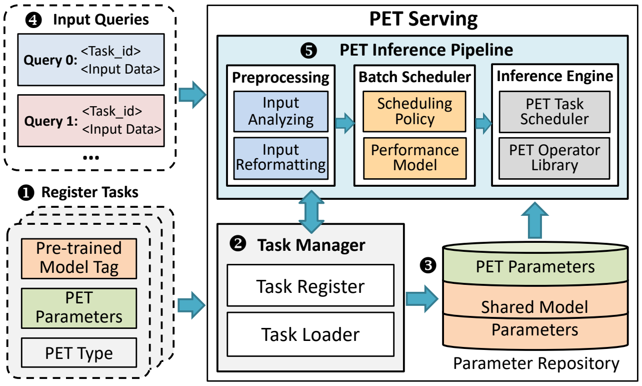


Figure 4: PetS System Overview.

3.3 Managing PET Tasks

One of the key features of PetS is the flexible and efficient PET task management mechanism, which is powered by the Task Register and Task Loader modules.

Task Register: The Task Register module registers a PET task according to the user-provided information denoting its shared model, task-specific parameters, and PET type. A triplet <`Task_id, Shared_model_tag, PET_type`> is formed to bind each task with its corresponding pre-trained model and the supported PET type, where the `Task_id` is unique to identify each PET task. All these triplets are organized as a map structure, with the `Task_id` as the key and the <`Shared_model_tag, PET_type`> pair as the value. Therefore, we can index the metadata of each task given the `Task_id` of a query. Once a PET task completes registration, its PET parameters are stored in the Parameter Repository. Note that for PETs like MaskBert and Diff-Pruning, the PET parameters are stored in a compressed format to save storage.

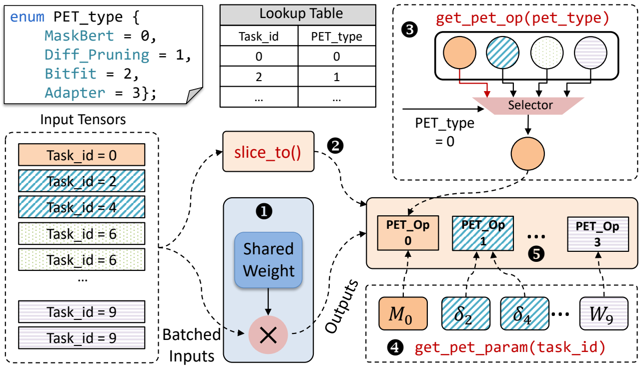
Task Loader: Before a PET task is invoked by the inference engine, the Task Loader module firstly loads the shared model parameters if they have not been loaded yet. Otherwise, the Task Loader indexes the Parameter Repository and accesses the PET parameters according to the `Task_id` of each invoked task. Considering that the PET parameters are small in size and the shared model only has one copy, all the parameters can be buffered into the GPU memory for quick invoking.

3.4 PET Inference Pipeline

At the core of PetS framework is the PET Inference Pipeline, which processes queries with three pipelined steps including Preprocessing, Batch Scheduling and PET Inference.

3.4.1 Preprocessing

The preprocessing module fetches queries from standard HTTP/gRPC data plane similar to conventional inference serving frameworks [18, 40]. Then it analyzes input data



```
enum PET_type {
    MaskBert = 0,
    Diff_Pruning = 1,
    Bitfit = 2,
    Adapter = 3};
```

Lookup Table

Task_id	PET_type
0	0
2	1
...	...

Figure 5: Base PET inference engine workflow.

and reformats them for the next query-batching step. Firstly, the input data is classified according to the shared model (Shared_model_tag). The metadata of each query is extracted, such as the invoked task's id, sequence length, PET type, etc. Some preliminary data preprocessing operations are then performed according to the extracted metadata, such as grouping the queries of the same PET task. In order to improve the performance of sparse PET operations, the input tensors' data layout is also reformatted according to the compression format of the corresponding sparse PET parameters. Finally, the preprocessed input queries together with the extracted metadata are dispatched to different queues for further scheduling, according to their targeting shared models.

3.4.2 Batch Scheduler

As discussed before, batching is an effective way to improve system throughput. Though PetS enables both inter-task and inter-algorithm batching through the proposed unified representation, the heterogeneity of queries in terms of PET type and sequence length still prevents batching efficiently. The batch scheduler module is used to overcome the challenge posed by query heterogeneity. Taking preprocessed queries as input, the batch scheduler tries to maximize the benefit of batching PET operators and minimize the padding overhead of batching shared operators with different sequence length at the same time. It leverages an accurate performance model to help make batching decisions. The details of the scheduling policy will be described in Section 4.1.

3.4.3 PET Inference Engine

PIE Workflow: The batched queries are finally fed into the PET Inference Engine (PIE). Figure 5 illustrates the base workflow of PIE. In the Figure, we use different colors to indicate the queries' PET types in the batch. For example, task 0 belongs to MaskBert, while tasks 2,4 belong to Diff-Pruning. PIE starts the computation of each Transformer layer as follows: ❶: PIE performs batched GEMM computation using the input tensor and shared weights W. ❷: PIE gets the PET_type attributes of each query by searching in the lookup table with its Task_id. The batched inputs are also sliced into several mini-batches (intra-task batching) according to the task id. Then PIE gets the PET operators (❸) and PET parameters (❹) according to the obtained PET types. ❺: PIE executes the PET operators successively on each sliced mini-batch. These PET operators are also responsible for adding the PET results to the shared outputs if needed. Note that the remaining operations like self-attention are also performed by PIE but are not shown in the figure.

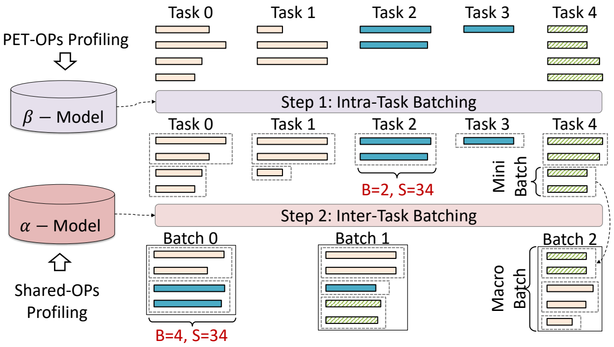


Figure 6: Coordinated Batching Strategy

PET Operator Library: According to the unified representation in Figure 3, the PET tasks rely on different PET operations. MaskBert and Diff-Pruning involve sparse matrix-matrix multiplication (SpMM) as their PET parameters contain high sparsity. Adapters perform light-weighted dense GEMM. While Bitfit only requires a Vadd operation. Therefore, PIE provides an operator library containing high-performance implementations of both dense and sparse operators. The sparse operators are tuned specifically for the sparse patterns and parameter sizes of the target models. We can also implement new PET operators for other emerging PET algorithms if they can fit into the unified representation.

PET Task Scheduler: During the inference of each layer, the PET operations of different tasks have no data dependency and can therefore run in parallel. Given the system allowed parallelism, e.g., the number of CUDA streams on GPU, the PET task scheduler schedules the PET operations to utilize the parallelism as much as possible. The PET operator scheduling strategy is introduced in Section 4.2.

4 Optimization Strategies

4.1 Coordinated Batching

In real scenarios, the input queries usually have variable sequence lengths. If we batch short queries and long queries, the short ones have to be zero-padded, incurring useless computation. Previous frameworks like TurboTransformers [12] pay much attention to solve such a problem. However, for PetS, we have to consider both the shared operations and PET operations. Therefore, we propose a Coordinated Batching (CB) strategy to coordinate these two parts during batching.

Problem Formulation: Assume there are R queries, namely $Q = \{x_0, x_1, ..., x_{R-1}\}$ associated with T different tasks. We divide queries into M batches. For each batch, we use $\alpha[N][L]$ to denote the shared model latency when batching N queries with a maximum length of L. In the meantime, a PET operator

Algorithm 1: Coordinated Batching Strategy

```
Input: Number of tasks $T$, queries $Q = \{x_0, x_1, ..., x_{R-1}\}$, Shared
  Op latency model $\alpha$, PET Op latency model $\beta$;
Step 0: Pre-processing
Cluster input queries to the same task and generate
  $Q = \{X_0, X_1, ..., X_{T-1}\}$, where $X_i$ contains $n_i$ queries;
Step 1: Intra-task batching
for $i \leftarrow 0$ until $T$ do
    Create DP state vector $state[n_i + 1]$, $state[0] = 0$;
    Sort queries in $X_i$ according to the sequence length in an
      ascending order;
    Create $split\_idx\_list[n_i + 1]$, $pt = get\_pet\_type(i)$ ;
    for $j \leftarrow 1$ to $n_i$; $min\_cost = INF$ do
        for $k \leftarrow 1$ to $j$ do
            $tmp = state[k-1] + \beta[pt][j-k+1][X_i[j].len]$);
            if $tmp < min\_cost$ then
                $min\_cost = tmp$, $split\_idx = k - 1$;
        $state[j] = min\_cost$, $split\_idx\_list[j] = split\_idx$;
    Split queries into mini-batches $MB$ using $split\_idx\_list$;
Step 2: Inter-task batching
Sort mini-batches according to their max sequence lengths;
Create DP state vector $state[\#mini\_batch + 1]$, $state[0] = 0$;
Create $sum[\#mini\_batch + 1]$, $sum[i]$ records the total queries of the
  first $i$ mini-batches;
for $i \leftarrow 1$ to $\#mini\_batch$; $min\_cost = INF$ do
    for $j \leftarrow 1$ to $i$ do
        $batch\_size$ = sum[i]-sum[j-1];
        $tmp = state[j-1] + \alpha[batch\_size][MB[i].max\_seq\_len]$);
        if $tmp < min\_cost$ then
            $min\_cost = tmp$, $split\_idx = j - 1$;
    $state[i] = min\_cost, split\_idx\_list[i] = split\_idx$;
Split mini-batches into macro batches using $split\_idx\_list$;
Return: The scheduled macro batches
```

Algorithm 2: PET Operator Scheduling Strategy

```
Input: PET operator set $\boldsymbol{O}$ of a macro-batch, stream set $\boldsymbol{S}$,
  latency model $\beta$ and bandwidth model $\omega$
Output: $\boldsymbol{O}$ to streams assignment $\Phi(\boldsymbol{O} \rightarrow \boldsymbol{S})$
$\boldsymbol{I} \leftarrow \emptyset$ ▷ operational intensity of $\boldsymbol{O}$;
for $o \in \mathbf{O}$ do
    $op\_intensity \leftarrow (\frac{o.FLOPs}{\beta(o)})/\omega(o)$;
    $\boldsymbol{I}$.append($op\_intensity$);
Sort $\boldsymbol{O}$ in an ascending order according to $\boldsymbol{I}$;
for $o \in \mathbf{O}$ do
    $stream\_idx \leftarrow \lfloor o.idx/|\boldsymbol{S}| \rfloor$;
    $\Phi(\boldsymbol{O}[o.idx]) \leftarrow \boldsymbol{S}[stream\_idx]$;
```

takes $\beta[pt][n][l]$ seconds to process the PET terms of n queries, whose `PET_type` is shortened to pt, and l is the max length of these n queries. Then, the estimated execution latency is:

这里可以多个 streams 进行进一步并行

$$Batch\_Latency(B_i) = \alpha[N_i][L_i] + \sum_{j=0}^{t_i-1} \beta[pt_{ij}][n_{ij}][l_{ij}]. \quad (2)$$

In the formula, we use B_i to denote the i-th batch, and assume there are t_i different tasks in the batch. For the j-th task in batch i, there are n_{ij} queries that shape a mini-batch, which is processed by a PET operator indexed by pt_{ij}. The longest sequence length of the n_{ij} queries is l_{ij}. Since all the M batches, namely $\mathcal{B} = \{B_0, B_1, ..., B_{M-1}\}$ are executed successively, we can further estimate the total latency as:

这里也是两个 level 的 batching

$$\begin{aligned} Total\_Latency(\mathcal{B}) &= \sum_{i=0}^{M-1} Batch\_Latency(B_i) \\ &= \sum_{i=0}^{M-1} \alpha[N_i][L_i] + \sum_{i=0}^{M-1} \sum_{j=0}^{t_i-1} \beta[pt_{ij}][n_{ij}][l_{ij}] \end{aligned} \quad (3)$$

相同 op 不同 seq_len, bs 对应的 lat 不同且不可建模估计，因此需要 DP 来决定 bs 和 max seq_len (即哪些 queries 组成一个 mini batch)

As we can see, the total latency is jointly determined by the shared and PET operators. To coordinate these two parts, we propose a two-step Coordinated Batching strategy. As illustrated in Figure 6, in the first step, we generate "mini-batches" for each task (intra-task batching) , which only considers the effect of batching PET operators using a profiled β-model. In the second step, we generate "macro-batches" by combining these mini-batches among tasks (inter-task batching), which only considers the effect of batching shared operators using a profiled α-model. In both steps, we sort the queries and use dynamic programming (DP) to find the optimal splitting positions with low time-complexity.

Algorithm 1 details this strategy. The queries with the same task id are firstly clustered and sorted according to the sequence length. At the first step, we use $state[i]$ to record the minimum latency of PET operations when batching the first i queries. We use $split\_idx\_list$ to record the splitting positions. Equation 4 shows the Bellman equation:

$$state[i] = \min_{0<j\leq i}(state[j-1] + \beta[pt_{ij}][i-j+1][l_{ij}]) \quad (4)$$

With the DP algorithm, we divide the queries of each task into mini-batches. At the second stage, the Bellman equation only considers the shared operators, whose latency is estimated by the α model. Instead of scheduling each single query, the second step schedules the mini-batches:

$$state[i] = \min_{0<j\leq i}(state[j-1] + \alpha[batch\_size][L_j]) \quad (5)$$

Where $state[i]$ records the minimum latency of batching the first i mini-batches. L_j denotes the max sequence length of the j-th mini-batch. $batch\_size$ denotes the number of total queries from mini-batches i to j. After dynamic programming, the mini-batches are assigned to multiple macro-batches.

4.2 PET Operator Scheduling

In addition to the coordinated batch scheduling, PET operators can be executed in parallel to further improve hardware utilization and performance. To achieve the PET-task-level parallelism on GPU, PET operators in a macro-batch (as shown in Figure 6) can be assigned to multiple CUDA streams. However, naïvely assigning a unique stream to each PET task may not get the ideal speedup, because if we assign computation-intensive operators to different streams (or memory-intensive operators), they can hardly be executed in parallel, since they are bounded by the same resources. Therefore, we propose a light-weighted online scheduling strategy to dynamically assign PET tasks to streams. The scheduling algorithm is shown in Algorithm 2. The input includes the set of PET operators to be scheduled, and the set of streams on which the PET operators execute. The algorithm also requires the PET latency model β used in Algorithm 1, as well as a bandwidth

Table 2: PET data structures and interfaces

Data Structure	Interface
`PETModel`	`load_shared_model(model_url)`
	`load_pet_task(pet_type,param_url)`
`PETLayer`	`load_pet_params(pet_type, pet_layer_param)`

model ω generated together with β. The algorithm outputs the assignment from the PET operator set to the stream set. The first step (lines 3–6) of Algorithm 2 computes the operational intensities of all the PET operators. Operational intensity is a metric to measure the compute to memory access ratio of an operator [57]. The achieved intensity of an operator is computed by its FLOPs divided by the utilized bandwidth. The second step (lines 7–10) then assigns a stream for each PET operator. The rationale is to put operators with differentiated operational intensities to different streams, in order to minimize resource conflict between streams.

Though the PET operator level parallelism contradicts the assumption that the PET operators are executed sequentially in Coordinated Batching. Experiments in Section 6 demonstrates that the coordinated batching still works well with parallel PET execution. Involving a more accurate performance model for parallel PET execution can help generate better scheduling results. We leave this as our future work.

5 Implementation

We implement `PetS` with a Python front-end to describe shared model and PET tasks management, and a C++ backend to perform query scheduling and inference serving.

5.1 PET Description

The description of PET tasks is based on the HuggingFace Transformers framework [58]. We extend HuggingFace Transformers library mainly with two data structures and three interfaces to manage PET tasks, as shown in Table 2. `PETModel` is the base structure to implement a model with PET tasks. `PETLayer` is defined in `PETModel` to describe PET operations, apart from the shared operations. The first interface of `PETModel` in Table 2, `load_shared_model`, loads the shared parameters as traditional Hugging Face tasks do. The `load_pet_task` interface is used to load PET parameters, given PET type and PET parameters URL. It will call `load_pet_params` defined in `PETLayer` to finish the underlying load operations for each layer. Users can inherit and implement these interfaces according to specific tasks.

5.2 Inference Serving

The three modules of the `PetS`'s PET Inference Pipeline in Figure 4 are deployed in individual processes to process input queries in a pipelined manner.

Inference Engine: Inference frameworks compatible with HuggingFace Transformers library can be plugged into `PetS` as its backend engine, such as TurboTransformers [12], LightSeq [56], FastTransformers [37], etc. Modifications should be done to support PET operators and the PET Task Scheduler. Without loss of generality, `PetS` implements the backend inference engine based on TurboTransformers [*]. It leverages cuBLAS to compute the shared dense MVM operators. We leverage a high-performance SpMM implementation [15] to implement the sparse PET operators.

Code Listing 1: User Interface

```
server = PetS() # create a PET server
# Register PET tasks
server.register_task("Adapter", "bert-base", pet_param_url_0)
server.register_task("MaskBert", "bert-base", pet_param_url_1)
# Register other PET tasks ...
# Load shared model parameters and PET tasks
server.load_shared_model("bert-base")
server.load_pet_tasks(pet_task_ids)
# Fetch queries from input query queue and run inference.
queries = server.fetch(input_query_queue)
 results = server.inference(queries)
```

Table 3: Shared Model Configurations

Bert Type	#Layer	#Head	Hidden Size	Inter-Size	# Params
DistillBert	6	12	768	3072	66 M
Bert-base	12	12	768	3072	110 M
Bert-large	24	16	1024	4096	340 M

5.3 User Interfaces

We use a code sample as shown in Code 1 to demonstrate how users can launch `PetS`, load models and process queries with only a few lines of Python code.

After creating a `PetS` server, it firstly registers PET tasks through the `register_task` interface. It will write the user-provided PET parameters to Parameter Repository and get the assigned task ids. Then the server loads the shared models by calling standard HuggingFace Transformers API and loads PET tasks using the assigned task ids to index the Parameter Repository. Once fetching a group of queries from the input query queue, the `PetS` server runs the PET inference pipeline and returns the inference results.

Currently, we only implement the four aforementioned PET algorithms in the `PetS` framework. A new PET algorithm can work with `PetS` as long as it meets two requirements: (1) The PET operations are separable (with necessary equivalent transformations) from the shared operations. (2) The separated PET operations are light-weighted. Then, to support a new algorithm, the developers should first identify its PET operations. Then the related functions introduced in this section should be extended accordingly.

6 Evaluation

6.1 Experimental Setup

Shared Models: We choose Bert-base, Bert-large [9], and DistillBert [49] as the shared pre-trained models, whose configurations are listed in Table 3. We do not include generative Transformer models such as GPT-2 [45] because GPT-like models have not been well-studied by the PET algorithms discussed above. We leave the evaluation on GPT-like models as our future work and focus on Bert-like models in this

[*] https://github.com/Tencent/TurboTransformers

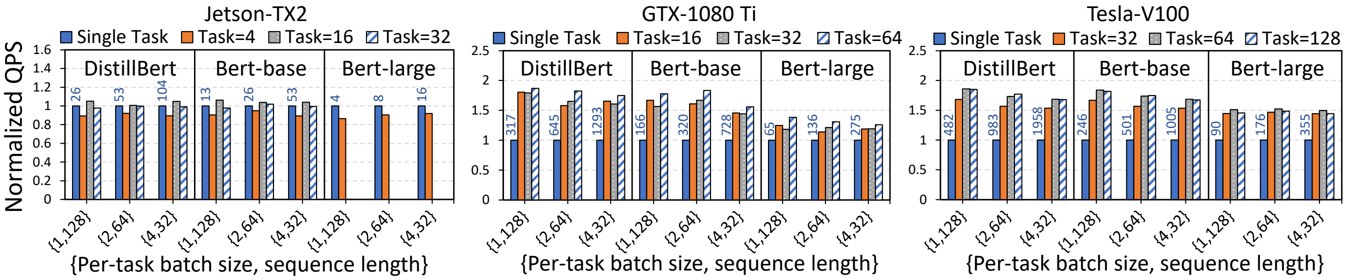


Figure 7: Throughput Improvement Evaluation on Multiple Platforms.

Table 4: PET Configurations

PET Type	Configuration	Main PET Params
Adapter	Bottleneck = 64	W_{down} and W_{up} at the `BertOutput`, `SelfAttenOutput` layers.
MaskBert	95% Sparsity*	Binary masks for all the linear weights.
Diff-Pruning	99.5% Sparsity	Sparse difference concerning linear weights, bias terms, and the classification layer.
Bitfit	N/A	Bias terms of linear / layernorm layers and a classification layer.

* Obtained through equivalent transformation

Table 5: Profiling of Total Supported Tasks

Platform	Device Memory	Shared Models	DistillBert	Bert-base	Bert-large
			SeqS / PetS	SeqS / PetS	SeqS / PetS
Jetson TX2	8GB*	Supported Tasks	34 / **504**	17 / **180**	3 / **12**
GTX-1080Ti	11GB		56 / **1336**	28 / **588**	7 / **126**
Tesla-V100	32GB		170 / **4344**	85 / **2164**	25 / **560**

* Shared by CPU and GPU

paper. Note that although in Section 2 we mainly use a single layer for illustration, the evaluations in this section are all conducted on entire models.

PET Tasks: The configurations of four PET algorithms are summarized in Table 4. For Adapter, we set the hidden size (d_m) of the adapter modules to 64. For MaskBert and Diff-Pruning the PET parameters' sparsity is set to 95% and 99.5%, respectively (we use the equivalent transformation proposed in Section 3.1 to obtain the 95% sparsity for MaskBert). For all PETs, we reproduce the algorithm on HuggingFace Transformers to obtain the trained PET parameters.

Platforms: We evaluate PetS on Edge/Desktop/Server platforms, namely Jetson TX2 (8GB memory, shared by CPU and GPU), GTX-1080Ti-11GB (Intel Xeon E5-2690 CPU), and Tesla-V100-32GB (Intel Xeon Golden 5220 CPU, two sockets). The V100 platform installs CUDA-10.1. The 1080 Ti platform installs CUDA-11.3. The TX2 platform is flashed with Jetpack 4.4.1 containing CUDA-10.2.

6.2 Main Results

6.2.1 Maximum Number of Supported Tasks

We first demonstrate PetS's scalability by comparing the maximum number of supported tasks with conventional Sequential Serving Systems (SeqS) [1, 7, 12]. Without loss of generality, here we use the unmodified TurboTransformers framework as a representative for SeqS. SeqS loads full-model copy for each task, while PetS works on light-weighted PET tasks. For each platform, we load T tasks (for PetS, each task belongs to a random PET type). If the system can process a batch of 32 randomly-generated queries (each query has a length of 128) without the out-of-memory (OOM) issue, we assume the system can support at least T tasks. We increase T repeatedly to test the limit. The maximum supported tasks are listed in Table 5. Compared to conventional SeqS systems, PetS supports 4× (Bert-large on TX2) to 26× (DistillBert on V100) more concurrent tasks, thanks to the proposed unified representation and efficient PET tasks managing mechanism. Therefore, PetS can substantially save the hardware cost when deploying multiple Transformer-based applications to scenarios from edge computing to cloud computing. Also, it avoids the notoriously slow model swapping [19, 48, 53] even when hundreds to thousands of tasks are invoked.

6.2.2 Throughput Improvement

As stated before, PetS achieves both inter-task and inter-algorithm batching through the unified representation and a specialized PET Inference Engine (PIE). Therefore, we evaluate PetS's throughput (measured in Queries-Per-Second, QPS) under different situations. As shown in Figure 7, we load 4~32,16~64 and 32~128 random tasks on TX2, 1080 Ti and V100 platforms, respectively. For each task, we generate queries with three fixed shapes. All queries with the same shape are executed in one batch. We adopt SeqS running a single task as the baseline. Note that here we do not include the two optimizations introduced in Section 4 and adopt a simple fixed-batch policy in the batch scheduling step. We assume that there are no dependencies between tasks.

As we can see, on the 1080 Ti and V100 platforms, PetS achieves up to 1.87× and 1.86× higher throughput, 1.53× and 1.63× on average, compared to the single-task serving baseline. We notice that PetS fail to achieve meaningful speedup than single-task serving on TX2. This is because TX2 only has limited computation resources (256 CUDA cores) and therefore can hardly get benefited from batched inference. Similarly, on 1080 Ti and V100, we observe lower speedup on Bert-large models than Bert-base/DistillBert. This is because Bert-large has a larger layer size (see Table 3), which saturates the GPUs' computation resources more easily, diminishing the benefits of batched inference.

6.2.3 Comparison with ParS

Apart from Sequential Serving Systems (SeqS), several previous serving systems are built to support concurrent execution of multiple tasks in parallel [18, 40, 48], which belong to the

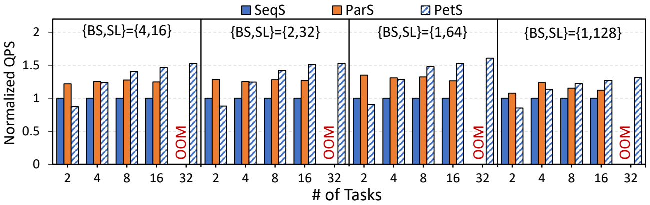


Figure 8: Throughput Comparison with SEQS and PARS.

Parallel Serving System (ParS). We compare PetS's throughput over conventional ParS to examine the performance. To implement ParS, we modify the original TurboTransformers framework and put each task-specific model to a unique CUDA stream to run all models in parallel. Note that in this experiment, we adopt Bert-base as the shared model and do not consider model swapping. All results are collected on the GTX-1080 Ti platform.

As shown in Figure 8, we evaluate multiple query configurations represented by a pair of per-task batch size and sequence length ($\{BS, SL\}$) to illustrate the PetS's generality. We run each query configuration on a different number of tasks for SeqS, ParS, and PetS. All results are normalized to the SeqS baseline. When the number of tasks is small (1~4), PetS cannot outperform ParS. For one thing, PetS has extra PET operations, as illustrated in Figure 3. For the other, tasks of ParS run in parallel. Although the shared weight part of PetS can also utilize parallel hardware, the overhead of PET operations cannot be offset by the limited parallelism. As the number of tasks increases, the benefit of PetS begins to manifest. PetS has an average 17.7% speedup over ParS when the number of tasks reaches 16 on all four configurations. Neither SeqS nor ParS could run too many concurrent tasks due to OOM, while PetS is still able to scale to 32 tasks and even more (refer to Table 5). As we can find in the figure, benefited from the higher hardware utilization, the QPS of PetS improves with the increased total batch size (i.e., #tasks× BS). As the number of tasks further increases to 256 or more (not shown in the figure), the QPS improvement curve will reach a plateau since a large batch saturates the GPU resources.

6.3 Performance Analysis

6.3.1 Execution Time Breakdown

To figure out why PetS outperforms the baseline systems in serving throughput, we break down the execution time of both PetS and SeqS on GTX-1080 Ti. We set two workloads (i.e., per-task batch size = 1, sequence length = 64 and per-task batch = 2, sequence length = 32) and evaluate eight random tasks with Bert-base and Bert-large models. Therefore, the two workloads issue 8 and 16 queries each time. As we can see in Figure 9, PetS speeds up the Non-PET operators (including the attention operations and the computation of shared linear layers) by 2.17× to 3.28×, thanks to the batched execution of shared operators. Due to the adoption of SpMM library, the PET operators only take up 27.4% to 41.3% of the total execution time. Therefore, the end-to-end execution time is still much less than SeqS.

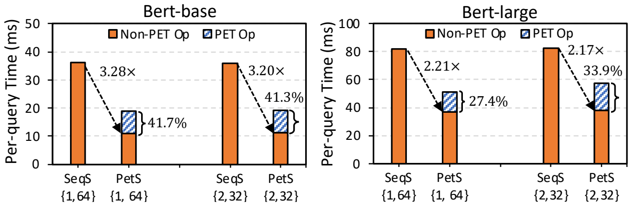


Figure 9: Execution Time Breakdown.

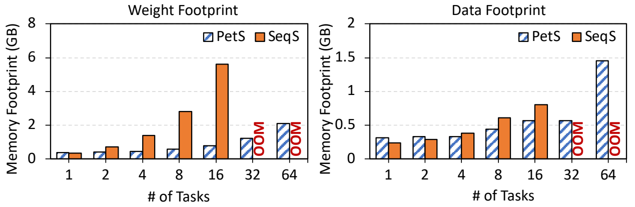


Figure 10: GPU Memory Footprint Comparison.

6.3.2 Memory Footprint Breakdown

We also profile the memory footprint of PetS and SeqS on 1080 Ti to understand why PetS has outstanding scalability. Taking the configuration of $\{BS, SL\} = \{1, 128\}$ as an example, we plot the consumed GPU memory by both model weights and data under different task numbers in Figure 10.

We can see that a single task has about 0.35GB weight parameters. The memory consumption of SeqS grows linearly with the number of tasks. The weight parameters exceed 11GB for 32 tasks on SeqS, causing OOM on GTX-1080 Ti. On the contrary, for PetS, only the memory footprint of the PET parameters, which normally occupies less than 5% of the shared weight, increases with the number of tasks. As a result, the total memory footprints of 64 tasks occupy less than 40% of total GPU memory, demonstrating that PetS can support much more tasks. Note that the memory footprints of 16 and 32 tasks are the same, but 64 tasks consume three times more data memory than 32 tasks. This is because we use NVIDIA CUB device memory allocator [36] for dynamic data memory management, and the allocated device memory is not strictly proportional to data size, but aligned according to some rounding rules.

6.3.3 Effect of PET Operator Scheduling

In Section 4.2, we introduce a PET Operator Scheduling strategy to properly schedule PET operators to multiple CUDA streams. To demonstrate the benefits of such a strategy, we also profile the speedup using Bert-base on GTX-1080 Ti GPU. We set the sequence length from 4 to 64. For each test case, we put the same 1024 random queries in the pool and process them with a batch of 128. As shown in Figure 11, when 32 tasks are served, increasing the number of streams to 32 brings the optimal performance for all input configurations and reduces up to 15% of execution latency. However, we observe that as the number of tasks keeps growing to 64 and 128, using more than two streams even downgrades the performance under the Seq $= 4$ configuration. The main inferred

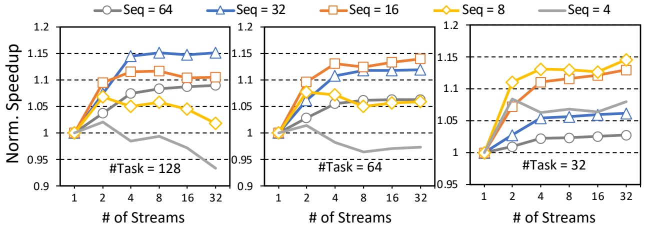


Figure 11: PET Operator Scheduling Performance.

reason is that, as the number of tasks increases, each task's batch size gets smaller. When the input queries also have limited sequence lengths (i.e., seq = 4), the execution time is too short for the GPU's scheduler to overlap concurrent streams. Thus, launching too many streams will only incur non-negligible synchronization overhead.

For PIE, the number of CUDA streams can be dynamically set before each batch's inference. Therefore, we can add a rule to the PET operator scheduler: for tasks with short sequence lengths and small per-task batch size, we set a small stream number such as one or two. For tasks with a long sequence length, we set a large stream number such as 32. We obtain the threshold on each GPU platform by profiling in advance.

6.4 Performance on Arbitrary Inputs

As stated before, in real-world multi-task serving scenarios, the input queries usually have variable sequence lengths and PET types. Naïvely batching these queries may not bring the ideal throughput. Our proposed Coordinated Batching (CB) strategy is aimed to improve PetS's performance on arbitrary inputs by coordinating shared operations and PET operations during batching. To evaluate the effect of CB, we test on workloads with variable sequence lengths and PET types, and then compare CB with three baseline batching strategies:

Fixed-Sized Batching: We put queries in the pool to fix-sized batches, regardless of their PET types and sequence lengths.

α-only Batching: We dynamically batch the queries only using the α model. This strategy is similar to TurboTransformers' smart batching. To implement the α-only Batching, we treat every single query as a mini-batch and only conduct the inter-task batching (step 2) in Figure 6.

β-only Batching: We dynamically batch the queries only using the β model. That is to say, in Figure 6, only the first step will be performed. The obtained mini-batches will be directly sent to PIE for execution.

To simulate real-world cases, we assume that the queries' lengths obey the Gaussian Distribution. Without loss of generality, we set the mean value to 32 and set the standard deviation from 1 to 8. The concurrent tasks are set from 32 to 128. Each task is assigned to a random PET type. For each case, we put 1024 queries in the query pool. For each query in the pool, we randomly assign it to a registered task.

As shown in Figure 12, the proposed CB strategy achieves on average 1.52× and 1.27× speedup over Fixed-Sized Batching and β-only Batching, respectively. When the std values are set from 1 to 4, CB also achieves up to 1.14× (1.06× on average) speedup over α-only Batching. For input queries with large variance, Coordinated Batching achieves lower QPS than α-only Batching. We infer that the first step guided by the β model may put some queries with certain length difference into one batch, which is acceptable by Step 1. Such a difference, however, may be amplified in Step 2, since the shared operators usually take up the majority of total execution time (see Figure 9). On the contrary, if the input queries have middle or small variances, the batching of both the two steps is near-optimal. Therefore, to get the highest performance on arbitrary inputs, we can measure the std value of the queries in the pool and choose to use Coordinated Batching (for low-std inputs) or α-only Batching (for high-std inputs).

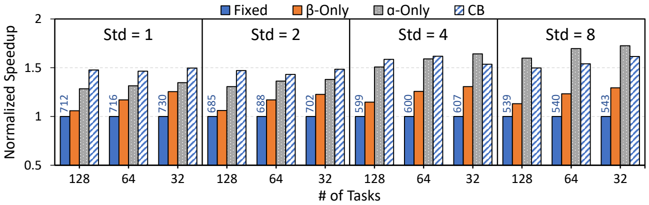


Figure 12: Comparison of Batching Strategies

7 Limitations & Future Work

As revealed by the evaluation results, PetS favors the scenarios where the number of tasks is large, while each task has few input queries. If there are only a few tasks, but each task has a large batch, using a traditional SeqS framework may also achieve good throughput. Also, as shown in Figures 8 and 9, when there are only a few tasks, the overhead of PetS (i.e, computing PET operators and shared dense operators separately) will outweigh the benefits, since the shared operators cannot achieve enough speedup in these cases to cover the overhead of PET operators.

Currently, the proof-of-concept PetS implementation only supports downstream tasks sharing the same pre-trained model. However, with the increasing of tasks adopting transformers as their backbones, there will be more and more shared models registered in PetS. The ability to simultaneously serve multiple pre-trained models from various task fields is needed. On the other hand, giant pre-trained models with trillions of parameters exceeding the capability of a single GPU have emerged to achieve significant accuracy gain, such as [13, 52, 63]. Partitioning a single giant model with PET tasks and different shared models on multiple GPUs is challenging for PetS. Moreover, for online queries with latency and cost constraints, how to balance PetS 's performance and QoS should also be taken into consideration. We leave the model partitioning and QoS-aware query scheduling as PetS's future work.

8 Related Work

Parameter-Efficient Transformers: Apart from the four representative PETs discussed in the paper, there are many other PET variants like AdapterFusion [41], LoRA [24], LeTs [14] and Prefix-tuning [30], etc. As stated in Section 5.3, these PETs can also work with PetS with some necessary extensions. We also notice that there are some works trying to put PETs into one framework such as UniPELT [33] and MAM-Adapter [21]. However, they belong to training frameworks designed from the user's angle. Their main goal is to combine multiple PET techniques into one model to achieve better accuracy. On the contrary, PetS is designed from a service provider's angle. It batches different fine-tuned tasks provided by users regardless of their PET algorithms and parameters. Therefore, these techniques are orthogonal to PetS. Besides, we also notice that Adapter-Hub [42] shares many ideas with PetS. It provides an easy way to train and share adapters for different downstream tasks based on Huggingface Transformers. A recent work named OpenDelta [10] further supports fine-tuning more types of PETs. However, the two frameworks are mainly designed for algorithm developers and not optimized for model serving. PetS mainly focuses on improving the multi-task serving efficiency from the system implementation and optimization perspectives. We believe that it will be promising to adopt PetS as the inference serving backend of these training frameworks.

Inference Serving Systems: As illustrated in Section 6, previous inference serving systems can be classified as SeqS and ParS. Rafiqi [53] and Clipper [7] deploy a model in an exclusive container, and introduce caching, batching, and model selection techniques to reduce model swapping overhead. Clockwork [19] reduces GPU inference latency variability by ordering queries based on their service level objectives (SLOs) and only running one query at a time, while TurboTransformers [12] batches queries to a single model to improve system throughput. Compared to the SeqS running each model sequentially, ParS systems enable concurrent execution of multiple models. INFaaS [48] proposes to automatically select models for multiple queries in order to maximize throughput. NVIDIA's MPS [39] and recent MIG [38] techniques enable efficient GPU resource sharing through hardware partition or full isolation. There exist other systems [8, 40, 60] featured with GPU sharing techniques.

Transformers Inference Engines: With the prevailing of Transformers, some inference engines are designed specifically for efficient Transformer inference. FastTransformers [37] and DeepSpeed [35] are two frameworks featured with multi-GPU inference. LightSeq [56] is a light-weighted inference engine performing some input-aware optimization techniques, such as smart batching and padding minimization, so does the inference engine of TurboTransformers [12].

Leveraging parameter-efficient Transformers, PetS saves storage, mitigates model swapping overhead and also improves system throughput by co-design and co-optimization between inference serving system and inference engine.

9 Conclusion

This paper presents PetS, a unified framework for efficient multi-task Parameter-Efficient Transformers (PETs) serving. To enable flexible PET task management and high-throughput serving, we first propose a unified representation to put different PETs into the same framework. Then we design a specialized PET inference engine to execute different tasks' queries in batches. We also propose a coordinated batching strategy to deal with arbitrary input queries and develop a PET operator scheduling strategy to exploit parallelism between PET tasks. Experiments on Edge/Desktop/Server GPUs demonstrate that PetS can support up to 26× more concurrent tasks and improves the serving throughput by 1.53× and 1.63× on Desktop and Server GPUs, respectively.

Acknowledgment

We thank all the reviewers and the shepherd for their valuable suggestions. This work is supported by NSF of China (61832020, 62032001, 92064006), Beijing Academy of Artificial Intelligence (BAAI), and 111 Project (B18001)

References

[1] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[2] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. A Multi-neural Network Acceleration Architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 940–953. IEEE, 2020.

[3] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language Models are Few-shot Learners. *arXiv preprint arXiv:2005.14165*, 2020.

[4] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-End Object Detection with Transformers. *arXiv preprint arXiv:2005.12872*, 2020.

[5] Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. The Lottery Ticket Hypothesis for Pre-trained Bert Networks. *arXiv preprint arXiv:2007.12223*, 2020.

[6] Yujeong Choi and Minsoo Rhu. Prema: A Predictive Multi-task Scheduling Algorithm for Preemptible Neural Processing Units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 220–233. IEEE, 2020.

[7] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 613–627. USENIX Association, 2017.

[8] Abdul Dakkak, Cheng Li, Simon Garcia De Gonzalo, Jinjun Xiong, and Wen-Mei W. Hwu. TrIMS: Transparent and Isolated Model Sharing for Low Latency Deep LearningInference in Function as a Service Environments. *CoRR*, abs/1811.09732, 2018.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[10] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. *arXiv preprint arXiv:2203.06904*, 2022.

[11] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *arXiv preprint arXiv:2010.11929*, 2020.

[12] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. TurboTransformers: An Efficient GPU Serving System for Transformer Models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.

[13] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *CoRR*, abs/2101.03961, 2021.

[14] Cheng Fu, Hanxian Huang, Xinyun Chen, Yuandong Tian, and Jishen Zhao. Learn-to-Share: A Hardware-friendly Transfer Learning Framework Exploiting Computation and Parameter Sharing. In *International Conference on Machine Learning*, pages 3469–3479. PMLR, 2021.

[15] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU Kernels for Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2020.

[16] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low Latency RNN Inference with Cellular Batching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys, pages 1–15, 2018.

[17] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Reddy Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, et al. Planaria: Dynamic Architecture Fission for Spatial Multi-tenant Acceleration of Deep Neural Networks. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 681–697, 2020.

[18] Google. TensorFlow Serving. https://github.com/tensorflow/serving, 2021.

[19] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 443–462, 2020.

[20] Demi Guo, Alexander M Rush, and Yoon Kim. Parameter-Efficient Transfer Learning with Diff Pruning. *arXiv preprint arXiv:2012.07463*, 2020.

[21] Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. Towards a Unified View of Parameter-Efficient Transfer Learning. *CoRR*, abs/2110.04366, 2021.

[22] Dan Hendrycks and Kevin Gimpel. Gaussian Error Linear Units (GELUs). *arXiv preprint arXiv:1606.08415*, 2016.

[23] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

[24] Edward Hu, Yelong Shen, Phil Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models, 2021.

[25] Yifan Jiang, Shiyu Chang, and Zhangyang Wang. Transgan: Two Transformers can Make One Strong Gan. *arXiv preprint arXiv:2102.07074*, 1(3), 2021.

[26] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. Ten Lessons from Three Generations Shaped Google's TPUv4i: Industrial Product. In *2021 ACM/IEEE 48th*

Annual International Symposium on Computer Architecture (ISCA), pages 1–14. IEEE, 2021.

[27] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. Heterogeneous Dataflow Accelerators for Multi-DNN Workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 71–83, 2021.

[28] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A Lite Bert for Self-supervised Learning of Language Representations. *arXiv preprint arXiv:1909.11942*, 2019.

[29] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 7871–7880. Association for Computational Linguistics, 2020.

[30] Xiang Lisa Li and Percy Liang. Prefix-Tuning: Optimizing Continuous Prompts for Generation. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 4582–4597. Association for Computational Linguistics, 2021.

[31] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A Robustly Optimized Bert Pretraining Approach. *arXiv preprint arXiv:1907.11692*, 2019.

[32] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin Transformer: Hierarchical Vision Transformer Using Shifted Windows. *arXiv preprint arXiv:2103.14030*, 2021.

[33] Yuning Mao, Lambert Mathias, Rui Hou, Amjad Almahairi, Hao Ma, Jiawei Han, Wen-tau Yih, and Madian Khabsa. UniPELT: A Unified Framework for Parameter-Efficient Language Model Tuning. *CoRR*, abs/2110.07577, 2021.

[34] Daniel Mendoza, Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. Interference-Aware Scheduling for Inference Serving. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, page 80–88, 2021.

[35] Microsoft. DeepSpeed for Inferencing Transformer based Models. `https://www.deepspeed.ai/tutorials/inference-tutorial/`, 2021.

[36] NVIDIA. CUB. `https://nvlabs.github.io/cub/structcub_1_1_caching_device_allocator.html`.

[37] NVIDIA. Fast Transformer. `https://github.com/NVIDIA/FasterTransformer`.

[38] NVIDIA. MIG. `https://docs.nvidia.com/datacenter/tesla/mig-user-guide/`, 2021.

[39] NVIDIA. MPS. `https://docs.nvidia.com/deploy/mps/index.html`, 2021.

[40] NVIDIA. Triton Inference Server. `https://developer.nvidia.com/nvidia-triton-inference-server`, 2021.

[41] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. Adapterfusion: Non-destructive Task Composition for Transfer Learning. *arXiv preprint arXiv:2005.00247*, 2020.

[42] Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. AdapterHub: A framework for adapting transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 46–54, Online, October 2020. Association for Computational Linguistics.

[43] Sai Prasanna, Anna Rogers, and Anna Rumshisky. When Bert Plays the Lottery, All Tickets are Winning. *arXiv preprint arXiv:2005.00561*, 2020.

[44] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. Pre-trained Models for Natural Language Processing: A Survey. *Science China Technological Sciences*, pages 1–26, 2020.

[45] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI blog*, 1(8):9, 2019.

[46] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[47] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. A Primer in Bertology: What We Know about How Bert Works. *Transactions of the Association for Computational Linguistics*, 8:842–866, 2020.

[48] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated Model-less Inference Serving. In *Proceedings of the 2021 USENIX Annual Technical Conference*, pages 397–411, 2021.

[49] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[50] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A GPU Cluster Engine for Accelerating DNN-based Video Analysis. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 322–337. ACM, 2019.

[51] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[52] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-billion Parameter Language Models using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[53] Sammy Sidhu, Jordon Wing, and Aakash Japi. Rafiqi: A GPU-Based Deep Learning Model Serving System. Technical report, University of California, Berkeley, 2020.

[54] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. GLUE: A Multi-task Benchmark and Analysis Platform for Natural Language Understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[55] Wenhai Wang, Enze Xie, Xiang Li, Deng-Ping Fan, Kaitao Song, Ding Liang, Tong Lu, Ping Luo, and Ling Shao. Pyramid Vision Transformer: A Versatile Backbone for Dense Prediction without Convolutions. *arXiv preprint arXiv:2102.12122*, 2021.

[56] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. LightSeq: A High Performance Inference Library for Transformers. *arXiv preprint arXiv:2010.13887*, 2020.

[57] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, 2009.

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

[59] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized Autoregressive Pretraining for Language Understanding. In *Advances in neural information processing systems*, pages 5753–5763, 2019.

[60] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. *CoRR*, abs/1902.04610, 2019.

[61] Li Yuan, Yunpeng Chen, Tao Wang, Weihao Yu, Yujun Shi, Zihang Jiang, Francis EH Tay, Jiashi Feng, and Shuicheng Yan. Tokens-to-token vit: Training Vision Transformers from Scratch on Imagenet. *arXiv preprint arXiv:2101.11986*, 2021.

[62] Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models. *arXiv preprint arXiv:2106.10199*, 2021.

[63] Wei Zeng, Xiaozhe Ren, Teng Su, Hui Wang, Yi Liao, Zhiwei Wang, Xin Jiang, ZhenZhang Yang, Kaisheng Wang, Xiaoda Zhang, Chen Li, Ziyan Gong, Yifan Yao, Xinjing Huang, Jun Wang, Jianfeng Yu, Qi Guo, Yue Yu, Yan Zhang, Jin Wang, Hengtao Tao, Dasen Yan, Zexuan Yi, Fang Peng, Fangqing Jiang, Han Zhang, Lingfeng Deng, Yehong Zhang, Zhe Lin, Chao Zhang, Shaojie Zhang, Mingyue Guo, Shanzhi Gu, Gaojun Fan, Yaowei Wang, Xuefeng Jin, Qun Liu, and Yonghong Tian. PanGu-α: Large-scale Autoregressive Pretrained Chinese Language Models with Auto-parallel Computation. *CoRR*, abs/2104.12369, 2021.

[64] Mengjie Zhao, Tao Lin, Fei Mi, Martin Jaggi, and Hinrich Schütze. Masking as an Efficient Alternative to Finetuning for Pretrained Language Models. *arXiv preprint arXiv:2004.12406*, 2020.

A Artifact Appendix

Abstract

The artifact contains PetS's code and its setup&running descriptions. We provide instructions and click-to-run scripts for reproducing the main results in this paper.

Scope

This artifact is used for reproducing the main results in Section 6. Specifically, we produce click-to-run scripts to reproduce the results of Figures 7,8,9,11,12 and Table 4.

Contents

- **Code Base:** The code base of the artifact includes the `PetS` inference framework. It contains the coordinated batching and PET operator scheduling components to demonstrate `PetS`'s performance optimization strategies.

- **Benchmarking Scripts:** We provide click-to-run benchmarking scripts to evaluate `PetS`'s performance. The script `run_pets_main_results.sh` can conduct all the main experiments, while you can also run each experiment individually using other provided scripts like `eval_batching_strategies.sh`, `eval_multi_stream.sh`, etc.

- **Instructions:** We provide a detailed README to guide the environment setup, evaluation and code reuse, etc.

- **Reference Results:** We provide the experiment results on two GPU platforms for reference use.

Hosting

The [†]artifact is archived in Zenodo.

Requirements

- **Hardware:** The artifact can run on a server or embedded platform equipped with at least one NVIDIA GPU. We tested NVIDIA Jetson TX2, GeForce GTX 1080 Ti, and NVIDIA Tesla V100.

- **Compilation and Runtime:** The experiments are performed on three platforms with GPUs mentioned above. The compilers and operating systems on the platforms are: on platform with 1080 Ti GPU, g++ 7.5.0 and nvcc 11.3, Ubuntu 20.04; Tx2: Jetpack 4.4.1 with CUDA-10.2. V100: Ubuntu 18.04, CUDA-10.1. g++ 7.5.0.

  Many other GPU platforms (e.g., 2080Ti, P100, K80, etc.) may also be compatible with this artifact. However, the Ampere architecture (e.g., A100, A6000) is not currently supported by the sputnik library.

Evaluation and Expected Results

After setting up the environment, you can run the two-step evaluation procedure: experiments running and results validating. The first step generates the performance numbers, and the second step draws figures. Please refer to `README.md` for detailed evaluation flow. The full evaluation lasts for about one hour. You can find the plotted results in the `research/reproduced_figures` folder.

Known Issues: Some AE reviewers have reported that in their running environments with V100 GPUs, they failed to get the same curve as Figure 11 (though Figure 11 is based on the 1080-ti GPU). We tested two machines with V100 GPUs. One can get even better results (a local machine, driver version = 510), but the other (an AliCloud instance, driver version = 460) got worse results than 1080-ti. We infer that this is due to the hardware and driver differences. The exact cause is still under investigation.

How to Reuse Beyond Paper

A PET algorithm can work with PetS as long as it meets two requirements:

- Its PET operations are separable (with necessary equivalent-transformations) from the shared operations.

- The separated PET operations are light-weighted.

To support a new algorithm, we should first identify its PET operations. Then three steps are required to add the new PET algorithm to PetS:

- Step-1. Register a new PET type and implement the PET operations using Pytorch APIs in `python/turbo_transformers/layers/modeling_pets.py`.

- Step-2. Deal with the PET parameters loading. Add new loading functions in `modeling_shared_bert.py` and `pet_manager.h`, respectively.

- Step-3. Implement the new PET operators in `shadow_op.cpp/shadow_op.h`

- Step-4. If the new PET operators should be called at places that are different from the four PETs in the paper, you should also modify the bert layers backends, e.g., `bert_output.cpp`

[†] https://doi.org/10.5281/zenodo.6534753