PredTOP: Latency Predictor Utilizing DAG Transformers for Distributed Deep Learning Training with Operator Parallelism

Dipak Acharya
Department of Computer Science and Engineering
University of North Texas
Denton, TX, USA
dipakacharya@my.unt.edu

Tong Shu*
Department of Computer Science and Engineering
University of North Texas
Denton, TX, USA
tong.shu@unt.edu

***Abstract*—With the increasing sizes of deep learning (DL) models, distributed training with various parallelization techniques, such as pipeline, model, and tensor parallelism, has become an absolute necessity. Accurately and efficiently predicting iteration latency for distributed DL training is critical for high-quality application scheduling, resource allocation, and automatic performance optimization. Unfortunately, existing latency predictors for distributed DL training mainly consider data parallelism, but fail to be applied for DL training optimized by other parallelization techniques. Also, the latency prediction of large-scale distributed DL training with hybrid parallelism is very challenging due to the extremely high measurement cost for one-time execution and the prohibitively huge search space of combinations from multiple parallelization techniques. In this paper, we propose PredTOP, a framework to accurately and efficiently predict the iteration latency of distributed DL training with different types of parallelization techniques. First, we numerically formulate the end-to-end time of pipeline parallelism using stage latency to leverage the low cost of white-box modeling. Then, we adopt a black-box modeling technique to predict the latency of each stage in pipeline parallelism. Specifically, we utilize Transformers over directed acyclic graphs to improve prediction accuracy. The performance superiority of our PredTOP is validated by extensive experimental results with two real-world benchmarks in comparison with state-of-the-art approaches and illustrated in a realistic automatic parallelization plan generation system.**

***Index Terms*—latency prediction, distributed deep learning training, intra- and inter-operator parallelism**

I. Introduction

The recent advancements in deep learning (DL) [15], [21]–[23], [41] have been primarily attributed to the increase in model size. The parameter size for state-of-the-art models has reached billions and trillions [3]. With the significant increase in the model size, a single accelerator or a GPU has been proven inadequate for training these models such as the Large Language Models. The state-of-the-art training systems [27], [42] usually rely on clusters of accelerators which can be used to train these large models in parallel. With the increased complexity of multi-device training, generating a good execution plan in the cluster is crucial for fast and efficient training. To evaluate every execution plan in a distributed environment, running the entire model is very expensive. Latency prediction in distributed DL training offers experts and automated systems valuable insights, enabling more efficient modeling of execution plans.

Several parallelization techniques have been applied to train DL models on distributed clusters. *Data parallelism* is a common technique that duplicates the model across several devices where each device operates only on a section of the data. *Pipeline parallelism* divides the model into sequential stages that run on different device meshes to form a pipeline. *Model parallelism* partitions the model into several parts and each device on the cluster operates on a part of the model. In *tensor parallelism*, each DL operator may be divided into multiple sub-problems and executed on multiple devices. DL training parallelism has also been explained as intra- and inter-operator parallelism. *Intra-operator parallelism* focuses on parallelism within a single operator. On the other hand, several operators are executed at the same time on different devices for *inter-operator parallelism*. With multiple parallelization techniques in DL training, the overall search space for a parallelization plan is extremely large which makes profiling all the combinations impossible. Moreover, parallelization techniques cannot be ignored in latency prediction as they have a huge impact on the training latency. Any system for predicting the DL training latency should consider the effect of the parallelization techniques used.

Unfortunately, existing training latency prediction approaches cannot evaluate DL models with various types of parallelization techniques. Yang et al. proposed utilizing execution trace sliding windows to predict the latency of DL training [39]. However, this work does not consider any parallelism which is a very important cornerstone for modern DL training. In [38], Yang et al. present runtime performance prediction on distributed training. However, this technique only considers data parallel training. Other DL training latency

This research work is done in the Smart High-performance and Ubiquitous Systems (SHU'S) lab at the University of North Texas. Dipak Acharya is a Ph.D. student under Dr. Tong Shu's supervision.

$*$ Corresponding author: Tong Shu, Assistant Professor, Department of Computer Science and Engineering, University of North Texas, Denton, TX 76207, USA (ORCID: 0000-0001-8617-1772) prediction approaches [7], [40] fail to consider parallelization techniques in DL training.

It is critical to make up for the lack of latency prediction approaches for distributed DL training with various parallelization techniques. However, we face the following two challenges. The first challenge is predicting latency in a large search space, formed by a combination of different levels of parallelism. The large number of operators on DL models makes simple white-box modeling extremely difficult. The second challenge in latency prediction for distributed DL arises from the fact that a single latency measurement requires significant resource allocation. The computational resources dedicated to preparing the execution plan detract from the valuable time and resources that could otherwise be used for DL training.

To overcome these challenges, we design and implement a cost-efficient modeling framework, PredTOP[1], a latency Predictor for distributed DL Training with Operator Parallelism. In PredTOP, we propose two main techniques that can be applied to intelligent optimization systems for DL training. First, we propose a grey-box prediction technique that can incorporate white and black-box prediction techniques for different forms of parallelism. We appropriately partition hybrid parallelism into two levels, each of which is both accurately and cost-efficiently modellable with suitable techniques, and then make two modeling techniques work synergistically to predict the end-to-end latency of distributed DL training. For pipeline parallelism, a simple white-box mathematical model can be used to predict the overall latency, because this type of parallelism is less complicated and can be formulated by a mathematical model.

For fine-grained parallelism techniques, such as model and tensor parallelism, we use a black-box modeling technique to predict the optimal latency. The reason behind this is that these types of parallelism have very high complexity, so that a simple mathematical model cannot represent them. Thus, we propose a black-box modeling method based on Transformer over Directed Acyclic Graphs (DAGs) [18] (simply called "DAG Transformer") to predict latency for a stage of DL training with model and tensor parallelism. This removes the need of optimization, compilation and profiling for DL model latency measurement significantly and provides a scalable latency prediction solution.

In this paper, we have proposed a general latency prediction approach that can be applied to existing DL training systems. Our main contributions are summarized below.

- As far as we know, PredTOP is the first work for predicting latency on distributed DL training with hybrid operator-level parallelism.
- We numerically formulate the end-to-end latency of pipeline parallelism based on stage latency in white-box modeling and synergistically combine it with black-box prediction for the latency of each stage with model and tensor parallelism.

[1]The source code is available at https://github.com/SHUs-Lab/PredTOP

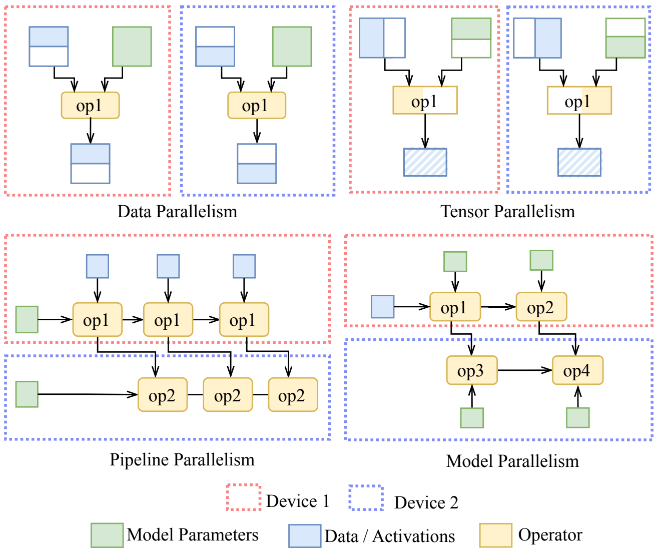


Fig. 1: Parallelization techniques in distributed DL training

- We improve the prediction accuracy of black-box modeling for DL training latency using the DAG Transformer prediction model.
- We integrate our PredTOP with a popular automatic parallelization system, Alpa [42], for distributed DL training and evaluate its performance with two real-world benchmarks. Experimental results show that our PredTOP can save the optimization cost for parallelization plan generation up to 46.6% at a negligible latency degradation of at most 2.1%.

II. Background and Motivation

A. Parallelism in Distributed Deep Learning Training

DL training relies on a sequence of computations with multi-dimensional tensors in the forward direction to compute the "activations" and backward direction to calculate the gradients followed by gradient update for the parameters. With large model sizes, currently available devices have insufficient memory to hold the model parameters. Moreover, training requires even more memory for several intermediate values, activations, and optimizer states. These require the models to be trained in a multi-device environment where different parallelization techniques need to be employed. We illustrate different parallelization techniques in Fig. 1.

Data Parallelism (DP): DP divides the data along the batch axis to multiple devices whereas the model itself is replicated on all devices. Each device only calculates part of the activations based on the section of the data. After gradient calculation, all the devices synchronize their parameter updates and apply them to their copy of the model. Memory optimizations for DP [26] allow training a model without replicating the entire model states in every device called *sharded DP*.

Pipeline Parallelism (PP): In PP [10], [14], [17], [20], [36], [37], the model is divided into sequential stages, each of which is assigned to one device to create a pipeline. The batch of data is further divided into multiple smaller microbatches and fed to the pipeline. This allows different devices to operate on different sections of the model with different microbatches at the same time.

Model Parallelism (MP): In MP, operators in a DL model are partitioned into multiple groups that are simultaneously trained on different devices by the same data samples. Here, operator groups are independent with each other for parallel execution, rather than dependent groups (i.e., stages) in pipeline parallelism.

Tensor Parallelism (TP): In TP, the model parameters of a single operator (e.g., matrix multiplication or self-attention) are sharded and distributed on different devices, each of which performs the forward computation with a part of model parameters. TP requires frequent synchronization with the all-reduce operations for both forward and backward computation.

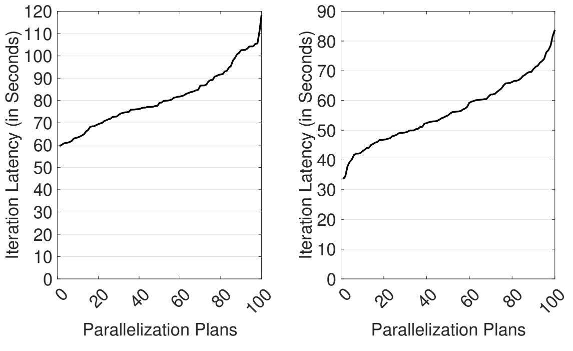
Parallelism for DL training can be classified into *intra-operator parallelism* and *inter-operator parallelism*. Intra-operator parallelism refers to parallelism within a single operator, such as tensor parallelism. Inter-operator parallelism is the parallelism between different operators in the computational graph of the DL model, such as model and pipeline parallelism. A parallelization plan consisting of different parallelization techniques can greatly optimize the execution performance of DL training.

While manual parallelization plans [27] are generated using rules based on expertise, they cannot be generalized to different models and device setups. Automatic parallelization such as Alpa [42], Galvatron [19], nnScaler [16], and FASOP [11], often rely on performing numerous profiling tasks to collect the runtime information for parallelization execution plan optimization. Here, one drawback is that the required profiling process is time-consuming.

B. Limitation of Pure Black-Box Modeling for DL Training

DL performance prediction requires the complex modeling of the DL model and the execution environment. DL performance modeling often uses the graph representation of the DL model to generate the performance metrics. Previous efforts such as Driple [38] and DNNPerf [7] used the graph representation of the DL model as input to a Graph Neural Network (GNN), such as a Graph Convolution Network (GCN), to predict the runtime computational performance, such as resource consumption and training latency. These approaches have seen success in a single-device execution environment. As larger models need to be trained in a distributed environment with multiple devices, this introduces further complexity, making performance modeling very difficult. This stems from the fact that the parallelization techniques used for DL training can substantially affect performance.

Single-device performance models have used the computational graph of a DL model as the input of a black-box model such as GNNs to predict the training latency. This principle breaks in multi-device environments, because a single model may exhibit varying performance based on the parallelization technique used in the distributed training. A single DL model running in the same execution environment, like Platform 2 (in §VII-A), can have multiple parallel execution plans, resulting in large variations in performance. Fig. 2 shows the variation in the execution latency of 100 parallelization execution plans for two different DL models. This shows that the same DL model and the hardware can have vastly different performances based on the parallelism plan selected.



(a) GPT benchmark (b) MoE benchmark

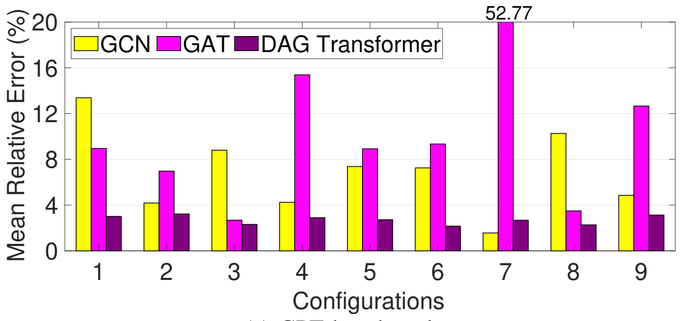
Fig. 2: Latencies of different parallelization plans

This variation in performance shows that any performance modeling of distributed DL training must take parallel execution plans into consideration. Furthermore, any simple black-box modeling technique is insufficient for predicting the performance as it needs to encode the information regarding the parallelization plan in addition to the DL model structure. Parts of parallelism types such as the *pipeline parallelism* can be modeled with a white-box model. On the other hand, other types of parallelism such as *operator parallelism* within a pipeline stage cannot be modeled with a simple white-box model at a low cost. The principle of black-box performance prediction can be used in distributed DL training for the intra-stage parallelism within the DL model stage being executed on each device mesh. Combining the results of smaller black-box models that predict the performance of model sections with a white-box model can accurately predict the overall execution performance of distributed DL training at an affordable cost.

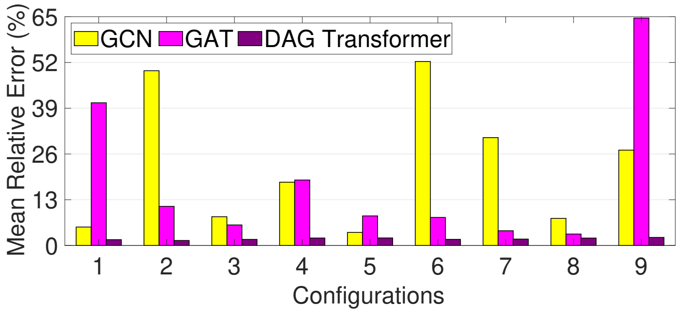
C. Inaccuracy of GNN-based Prediction for DL Latency

Message-passing GNNs have been popular for black-box performance modeling of DL models. These GNNs generate a vector representation of each node called *embeddings*. To do so, they start with the node features and update the embeddings by passing the node information to neighboring nodes on each GNN layer. While graph representation of DL can be done with layer-level operations such as *convolution* and *self attention*, this does not capture the complexity of implementation for these operations. Using the tensor level operators such as *matrix multiplication* and *element-wise add* does hold more information on the operator implementation, these types of graphs tend to be very large. Existing message-passing GNNs need a large number of layers to fully communicate between all nodes as the message is passed only between neighbors on each layer.

While existing GNN techniques, such as GCN and graph attention network (GAT), can be used to analyze any graphs, they fail to exploit the significant DAG property of the computational graph of DL models. Recent effort on GNNs [18]



(a) GPT benchmark



(b) MoE benchmark

Fig. 3: Stage prediction based on GCN and DAG Transformer

proposed using Transformers [34] for generating the graph embeddings on DAGs. This approach proposes using the attention mechanism and restricting it to only the predecessors and successors of any node for attention calculation on the nodes. These DAG Transformers can generate node embeddings of the graphs with very few layers by enabling communication between all relevant nodes. Fig. 3 shows the comparison of the prediction error of predicting execution latency for GCN and DAG Transformer for the same amount of training data. This shows that across different configurations, DAG Transformer shows consistently better prediction accuracy compared to GCNs.

III. Overview of Gray-Box Modeling Framework

Predicting the latency of distributed DL is very challenging due to hierarchical parallelism at intra- and inter-operator levels. Automated parallelization optimization systems face the challenge of navigating extremely large search space to find the optimal execution plan. While the combined search space of different parallelism techniques for distributed DL training is extremely complex, the search space for each type of parallelism is less complicated and can be modeled with different techniques.

We consider a scenario of training a DL model in a homogeneous cluster, where each computing node is equipped with multiple GPU devices. DL training can be accelerated by hybrid parallelism, including PP (with the highest scalability), MP, and TP (with the lowest scalability). To optimize the parallelization plan, it is better to organize PP, MP, and TP in a hierarchical way. In PP, each stage is compiled and executed on a mesh of devices that are close to each other. In each stage, operators are divided into one or more groups, each of which run in a subset of devices in one mesh. In an operator group, each operator might run in a single device or across multiple devices through TP. For each stage, the intra-stage compiler determines the optimal MP and TP plans to speed up the stage execution on the device mesh.

To better model distributed DL training, we first divide parallelism techniques into inter- and intra-stage parallelism. The former indicates PP among different stages, and the latter includes MP and TP within each stage. Then, we leverage different modeling techniques to tackle these two levels of parallelism. Inter-stage latency can be calculated very easily based on the execution latency of each stage. However, each stage execution latency depends on the intra-stage parallelization plan generated by the intra-stage compiler.

Correspondingly, we build a black-box predictor to estimate the optimal intra-stage execution latency of each stage on each mesh in the cluster. We only focus on the optimal latency generated by an intra-stage compiler because any DL training system will always use this optimal execution plan for the stage. With this predictor, we can quickly predict the optimal latency for all possible stages on all available meshes. In the next phase, we build a simple white-box mathematical model for inter-stage parallelism. Since we already have the latency of all possible stage and device combinations, the white box model can calculate the end-to-end training latency of the DL model at a very low cost.

IV. Black-Box Modeling for Stage Latency

In this section, we explain our approach for black-box modeling which can be applied to predicting intra-stage latency of each stage.

DL models are often represented as DAGs where the nodes are one of the various tensor operations such as matrix multiplication or element-wise ReLU operation. The edges in the DAGs represent the data dependencies between different operations of the models. Graph-based modeling for DL models has been proven extremely effective. Previous works have used different forms of Graph Neural Networks such as GATs [7] and GCNs [38] to predict various runtime information of DL models. The limitations of these types of models is that message passing between the nodes of the graphs require multiple stacked layers, as one layer only passes the node embeddings to its immediate neighbours. Because of this reason, the existing models use very high level DL model graphs, which do not provide enough details regarding the actual tensor level operations happening in them. While it is possible to generate large graphs with highly detailed operations such as the Jaxpr representation used by JAX [1], these graphs tend to be extremely large and infeasible to process with simple GNNs such as GCN and GATs. We observed that we could exploit the sequential nature of the DAG, using the transformer [34] architecture which can process very large sequential information in an extremely efficient manner.

A. *Transformers over Directed Acyclic Graphs*

Transformers [34] are extremely effective in processing sequential data such as texts by attending to the previous ele-

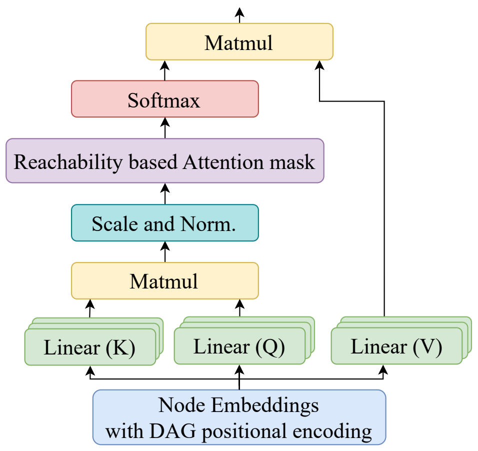


Fig. 4: Architecture of DAG Transformer Layer

ments in the sequence. Transformers over DAGs [18] applied this principle behind transformers in the context of DAGs. They do this by restricting the receptive fields of attention for each node to only those nodes that have a path between them. This means that a node n will attend to another node m if and only if there exists a path from m to n or vice versa. This same idea can also be applied to the DAG representation of the DL models to predict the execution latency of the models. The intuition behind this idea is that the overall latency will mostly be affected by the sequence of operators where every operator depends on one or more of the previous operators, which cannot run in parallel.

DAG Transformer [18] uses two main DAG-based properties to employ Transformers in the context of DAGs. The authors termed the first concept reachability-based attention (DAGRA). According to this, two nodes in a graph are more likely to be important to one another if they have a path between them. This implies that restricting the receptive fields of nodes to their predecessors and successors results in more effective attention. The second concept has been termed directed acyclic graph positional encodings (DAGPE), which incorporates the sequential nature of DAGs into the transformer model by using the depth of the nodes as the positional encoding for the transformer. For the implementation of DAG Transformers, the standard attention calculation proposed by Viswani et al. [34] can be modified using a mask matrix.

$$Attention(X) = softmax(\frac{QK^T}{\sqrt{d_K}} + M), \text{ where}$$
$$M(u,v) = \begin{cases} 0 & \text{if } u \in N_k(v) \\ -\infty & \text{otherwise} \end{cases} \tag{1}$$

Eqn. 1 shows the attention calculation on the DAG Transformer. In addition to the standard attention calculation, a mask matrix M is used which restricts the attention calculation to the successors and predecessors of the nodes. In addition, the constant k is also defined, which is a hyperparameter, that specifies the neighborhood range for attention calculation. In our case, this is set to ∞ as we want the attention calculation throughout the graph. The architecture of a DAG Transformer layer is shown in Fig. 4.

TABLE I: Node Parameters of stage DAG

Parameter Name	Description
Operator Type	One hot encoded vector representing the type of operator
Output Tensor Dimensions	Vector with the size of the output tensor dimensions
Output Data Type	One hot encoded vector for output data type
Node Type	One hot encoded vector for node type: input, literal, operator or output

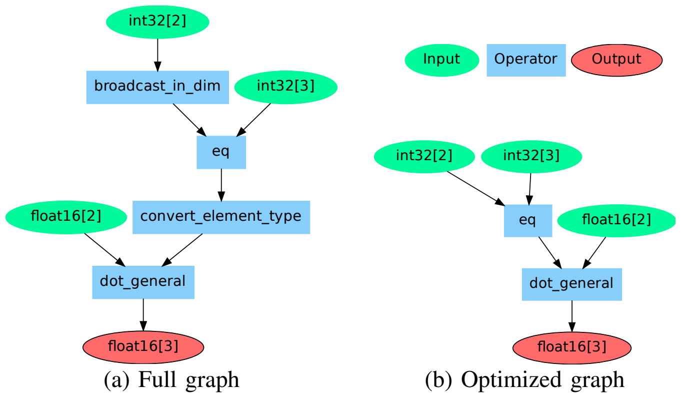


Fig. 5: Graphs constructed from the stages. *int32[n]* denotes a n-dimensional tensor of type *int32*

B. *Transformer-based Stage Latency Prediction*

1) Training Data: We collect the training data for our DAG Transformer model by randomly sampling the stages of the DL model. The latency for randomly sampled stages is collected by profiling. We use Alpa's [42] intra-operator optimizer to compile each stage for the optimal latency. We include the stages of different sizes to make our model more general.

2) DAG Construction: In order to input the model stage into the Transformer model, we first convert the stages into a DAG representation where each node is a tensor-level operation. The DL models are written using JAX [6]. We take the Jaxpr which is an internal representation used by JAX to build our graphs. Jaxpr consists of several equations which are tensor-level operations for the DL training. Each equation contains one or multiple inputs and outputs. We build directed acyclic graphs using these equations as the input for PredTOP. Fig. 5a shows an example of a simple graph constructed from the jaxpr representation of the stages.

3) Node Parameters: We select node parameters to encode enough information about the stages in the DAGs. Tbl. I shows the node features we select for training our predictor. These features sufficiently capture the information of the DL model. We focus mostly on data type and tensor size of the output tensor for each operator rather than input tensors, as input for every operator is always output from the preceding node. Thus, the data type and dimensions of all operator inputs will be recorded in the graphs. In addition to this, we apply logarithmic scaling for the tensor dimension. This is because tensor dimension is typically much larger than other features, potentially dominating the output.

4) Graph Pruning: The graph constructed from the jaxpr representation of the stages will often be very large. There are several operations whose effects can be stored in the graph even after removing them. These include operators such as *reshape* and *convert_element_type*. Every node will store the data type and tensor shape of the operator. If the data type is different between the two connected nodes. Then this will inherently imply that there was a data conversion between these nodes. We remove these redundant operators from our graphs to keep the graph size reasonable so that the prediction models can be trained more efficiently. Fig. 5 shows an example of graphs before and after pruning.

5) Model Design: Our latency prediction model consists of multiple DAG Transformer layers first to find the node-level embeddings of the DAG. Eqn. 1 shows the attention calculation for each layer in our model. After that, we use a pooling layer to get the overall graph embedding. For pooling we use the global add pool, which gives the graph-level embeddings by adding the node embeddings across the node dimensions. The intuition behind this is that the nodes in the graph of the DAG will have an additive effect on the overall latency of the stage.

$$x_i = \sum_{n=1}^{N_i} x_n^i \tag{2}$$

Eqn. 2 shows the global add pool function used by our model to calculate the graph level embeddings for i-th graph. Here, N_i is the number of nodes on the graph x_i is the graph embedding for the i-th graph and x_n^i is the node embedding of n-th node of i-th graph. After getting the graph embeddings, we use linear layers each with ReLU activation, and finally use the output layer for predicting the latency of the stages.

6) Hyperparameters: Through experimentation with different values for the hyperparameters, we found that the predictor performed best with 4 layers of DAG Transformer with the embedding dimensions of 64 for each layer. We also found that using a learning rate decay function gave us the best results. We use the cosine decay function for our learning rate, where the learning rate starts from 0.001 in the first epoch and reduces to 0 in the last epoch. We train the model for 500 epochs with a batch size of 32. For optimization we use the pytorch's Adam optimizer with default values of β_1=0.9 and β_2=0.999.

7) Loss function: We evaluated two loss functions, the Mean Squared Error (MSE) loss, and the Mean Absolute Error (MAE) loss for our model. We found the MAE loss function always outperformed the MSE loss. Thus, we selected the MAE loss function for our latency predictor.

$$L = \frac{\sum_{i=1}^{N} |\hat{y}_i - y_i|}{N} \tag{3}$$

Eqn. 3 shows the loss function used by our model. Here, N is the number of training samples, $\hat{y}_i$ and y_i are the predicted and true values for the latencies for the i-th training sample.

8) Early Stopping: To enable faster training of our predictor model, we employ early stopping. We record the weights of the best-performing model during training. If the validation loss of the model does not improve for 200 epochs while training, we stop training and reset the model to the best-performing model. This significantly reduces the training time for our prediction model and also improves accuracy.

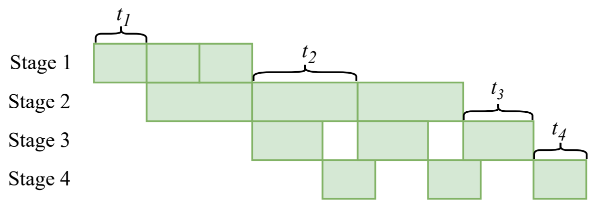


Fig. 6: Pipeline with four stages and three microbatches

V. White-box Modeling for Pipeline Parallelism

We use a simple mathematical model to incorporate the inter-stage parallelism model in PredTOP. Inter-operator parallelism is often far less complex compared to intra-operator parallelism. For pipeline parallelism, it is possible to evaluate the overall training latency based on the latency of each stage and the pipeline schedule [5], [10], [20] used. Alpa [42] used an inter-operator optimizer based on pipeline parallelism to optimize the parallel execution plan based on the '1f1b' [20] schedule. We use the same pipeline model in our latency prediction as well.

Pipeline execution for DL models splits the model into multiple sequential segments known as stages and executes these stages in a pipeline with a small segment of data referred to as a microbatch. Since the stages are sequential each stage execution depends on the output of the previous stage. For example, the execution of the i-th microbatch of the n-th stage depends on the output from the i-th microbatch of the $(n-1)$-th stage. Because of this reason, the stage with the highest execution time becomes the bottleneck for the overall pipeline execution.

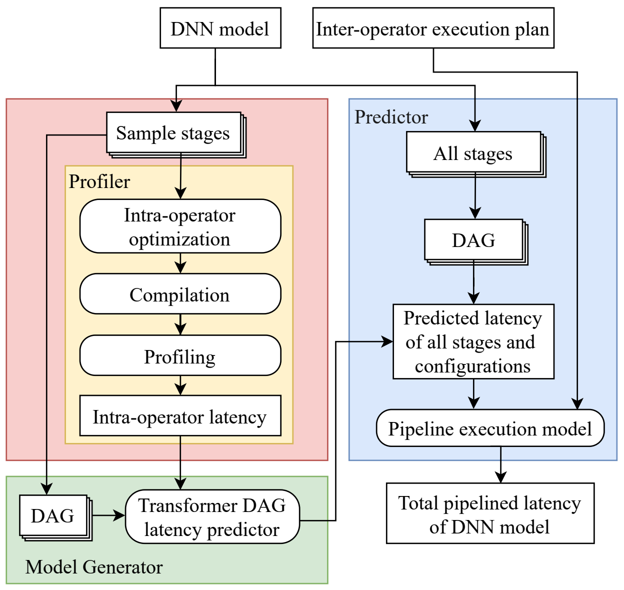
Fig. 6 shows the execution of a pipeline with four stages and three microbatches. In this example, Stage 2 has the longest execution time which creates a bottleneck for the pipeline execution. Eqn. 4 gives the execution time of the pipeline model.

$$T = \sum_{i=1}^{S} t_i + (B-1) \cdot \underset{1 \leq j \leq S}{max}\, t_j \tag{4}$$

Here, T is the pipeline execution time and t_i is the execution time of the i-th stage. S is the number of stages and B is the number of micro-batches in the pipeline. The total latency consists of two parts. The first part gives the sum of the execution latencies of all the stages in the pipeline. The second part represents the stage with the highest latency multiplied by the number of microbatches. In our white box model for pipeline parallelism, we ignore any communication time involved between the two stages. The reason is that in high bandwidth systems, the inter-stage communication time is negligible compared to the stage execution time.

VI. System Workflow

To put our latency prediction approach into practice, we implemented our approach in the prediction tool, PredTOP, and integrated it with Alpa [42] in view of its implementation of DL training parallelization to be most generalizable and inclusive of various parallelism techniques. Alpa is a compiler for distributed deep learning training that implements both



DNN model
Inter-operator execution plan
Sample stages
Profiler
Intra-operator optimization
Compilation
Profiling
Intra-operator latency
DAG
Transformer DAG latency predictor
Model Generator
Predictor
All stages
DAG
Predicted latency of all stages and configurations
Pipeline execution model
Total pipelined latency of DNN model

Fig. 7: System workflow

inter- and intra-operator parallelism to train the model fast and efficiently. Alpa achieves this by hierarchically optimizing each level of parallelism. It first iterates through all possible stages for pipelining and runs the intra-operator optimization pass on all the pipeline stages. Then, it profiles all these stages for collecting their runtime latency, which it will use to get the optimal execution plan.

While this approach works well, the problem is with the need to profile the large number of stages, which will greatly increase the optimization time. We implement PredTOP to alleviate this need for profiling. While Alpa runs the intra-stage optimization pass and profiles the runtime latency of all the possible stages, our approach only selects a subset of stages from all possible stages of the DL model. We then only profile these stages to collect the runtime latency of the stages. Using this profiled data, we train our predictor model based on the DAG Transformer to predict the optimal intra-stage execution time for all the stages.

Our implementation works on three main phases. They are highlighted in Fig. 7.

1) Profiling phase: In this phase, we first collect stages to profile by randomly sampling the stages of different sizes. We then run the intra-stage optimization provided by Alpa on these stages, compile them, and profile the selected stages on each mesh for collecting their runtime latency.

2) Training phase: In this phase, we take the stages selected for profiling in the profiling phase and build a DAG for each stage so that we can input these graphs into the predictor model. We use techniques discussed in §IV-B4 to preprocess the graph for training. Then, we take the intra-stage latency for each stage collected in the profiling phases and the graphs to train a DAG Transformer model for each mesh. The Transformer model used is discussed in more detail in §IV-B.

3) Prediction phase: In the final phase, we use the DAG Transformer model trained in the previous phase to make predictions for the runtime latency of all the stages on the corresponding mesh. For this phase, we select all the stages for prediction. Since we do not need to compile or profile the stages, the cost of latency calculation will be very low. After we get the latencies of all the stages on all different meshes we use the pipeline model to get the end-to-end latency of the DL training iteration.

VII. Experimental Setup

In this section, we implemented the proposed latency predictor system to evaluate its performance and effectiveness for real deep learning training systems.

A. Experimental Environments

We implemented our proposed predictor on the Alpa compiler [42]. Alpa works for DL models written in JAX library [6]. Our predictor based on DAG Transformer [18] is written using the PyTorch library [24]. The source code for the graph construction, training, and inference is all implemented in Python. Throught our experiments, we use python 3.9.20 with nvidia-driver 565.57.01.

We performed our experiments on two local setups.

- **Platform 1** is the Dell PowerEdge R750XA Rack Server has two 26-core/52-thread 2.2GHz Intel Xeon Gold 5320 CPU processors and 192 GB RDIMM main memory. The server is configured with two Nvidia Ampere A40 GPUs, connected via one Nvidia NVLink bridge with a bidirectional bandwidth of 112.5 GB/s. Each GPU has 10,752 CUDA cores, 48GB GDDR6 GPU memory, a memory bandwidth of 696 GB/s, and compute capability of 8.6.
- **Platform 2** is a local cluster with 2 Dell Precision 5820 Tower X-Series nodes. Both nodes have the intel core i9-10900X CPU with 10 cores, and 20 threads running on a 3.7 GHz base clock. Both of these nodes also have 64 GB of memory. Each of these nodes is equipped with two Nvidia RTX A5500 GPUs. These are the Nvidia GPUs from the Ampere Generation with 10,240 CUDA cores and 24GB GDDR6 memory. The GPUs within a node are connected with Nvidia NVLink bridge with bidirectional bandwidth of 112.5 GB/s whereas the two nodes are connected to each other with 10 GbE.

We experimented with different configurations from these two setup to form a mesh to execute each stage. We focus our experiments solely on homogeneous meshes, because DP and TP across heterogeneous devices are suboptimal, with one device inevitably becoming a bottleneck. The details of the five different mesh configurations we used are shown in table II. For each mesh, we also select the intra-operator parallelism configuration shown in table III. Each experiment is identified by (m, p) where m is the mesh index shown in table II and p is the parallelism configuration shown in table III.

B. Benchmarks

We focus on existing Transformer models as our benchmarks. We select two main models as our benchmark for

TABLE II: Mesh Configurations

Mesh Index	No. of Nodes	No. of GPUs per node
1	1	1
2	1	2
3	2	2

TABLE III: Benchmark Configurations

Mesh Index	Conf. Index	Remarks
1	1	Single GPU (No parallelism)
2	1	2 way Data parallel
2	2	2 way Model parallel
3	1	4 way Data parallel
3	2	2 way Data and 2 way Model parallel
3	3	4 way Model parallel only

prediction accuracy and optimization cost of our prediction approach.

- **Generative Pre-Trained Transformers Version 3 (GPT-3)** contains several stacked transformer layers. The specification of the GPT model used for the evaluation of our latency prediction system is shown in Tbl. IV.
- GShard **Mixture of Experts (MoE)** which has a mixed sparse and dense LMs. The specification of the MoE model used for the evaluation of our latency prediction system is shown in Tbl. IV.

TABLE IV: Benchmarks

	GPT-3	MoE
Number of Parameters	1.3B	2.6B
Sequence Length	1024	1024
Hidden layer size	2048	768
Number of Layers	24	32
Number of heads	32	16
Vocabulary size	51200	32000
Number of Experts	-	16
Experts group size	-	2048

C. Metrics

We used the following three metrics to evaluate the effectiveness of our approach.

- **Prediction Error** reflects how accurately our models will perform compared to the actual measurement of the latency by profiling the stages. We use the mean relative error (MRE) to evaluate the effectiveness of our prediction approach.
$$MRE = \frac{\sum_{i=1}^{N} \left| \frac{(\hat{y_i} - y_i)}{y_i} \right|}{N} \times 100\% \quad (5)$$
Here, N is the number of samples, y_i is the real value of the latency, and $\hat{y_i}$ is the predicted value of the latency. The smaller value of MRE represents a better prediction model.
- **Optimization Cost** gives the overall cost to formulate the optimal parallel execution plan using our approach. To formulate the optimal time, we use the Alpa [42] compiler. For the baseline value, we use the time to generate the optimal plan by profiling all the possible stages. With our approach, this includes the time for collection of the sample stage latencies, model training, and inference time.
- **Parallelism Plan Performance** denotes how well the execution plan generated by our approach will perform. We use the performance of the plan generated by full profiling as the baseline value against our approach. We use the Alpa [42] compiler for parallel plan generation.

D. Comparisons

To compare with applying the DAG Transformer into our PredTOP, we use following GNN models as comparisons.

- **Graph Convolution Networks (GCN)** [12] are simple message-passing GNNs working in a similar principle as Convolutional Neural Networks (CNNs). In these types of networks, the node information is shared between the neighboring nodes on each layer. We use the same node features as Tbl. I for this comparison with 6 GCN layers of size 256 each.
- **Graph Attention Networks (GAT)** [35] incorporate masked attention layers to attend to the features of the neighbouring nodes. Based on our experiments, we selected GAT model with hidden dimension of 32 and 6 layers, which we found as one of the best performing model configuration with comparable training time with our other baselines.

We run our latency prediction with Alpa [42] compiler to generate a parallel execution plan. The reason behind selecting Alpa is that among many existing parallel DL training optimizers, we found that Alpa provided the most general view of DL parallelism which incorporates DP, TP, and PP. For comparing the end-to-end performance, we compare the execution plan generated by Alpa using our prediction with Alpa's original profiling approach. Alpa tries to reduce the number of profiling tasks required for optimal parallel execution plan generation by reducing the stage-device imbalance in terms of the ratio of devices used for stage execution to the total number of devices and the stage size with respect to the model size. We also include this in the comparison with our approach. In addition, we also compare the total optimization time of Alpa with full profiling and our prediction model generation approach.

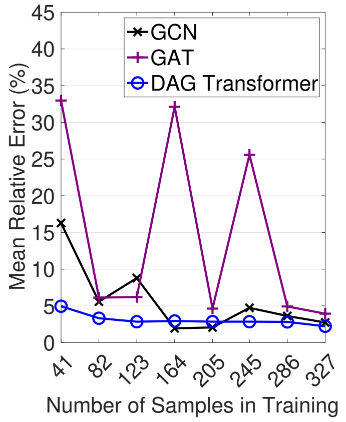
VIII. Evaluation

In this section, we evaluate the performance of the proposed latency prediction technique for distributed DL training with popular models and its efficacy for distributed DL execution plan generation.

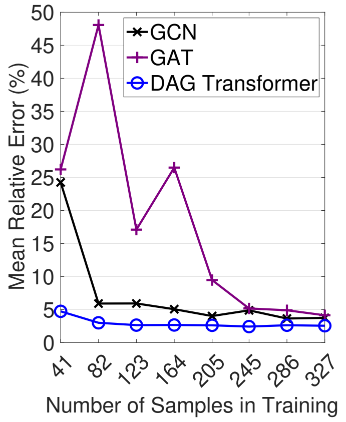
We measure the prediction error of our models compared to the true values of execution latencies measured by profiling stages on different runtime configurations. To perform the evaluation we collected 409 different stages from the GPT-3 benchmark and 205 different stages from the MoE model, and profiled the latency of all these stages on all runtime configurations.

A. Prediction Accuracy for Stage Latency

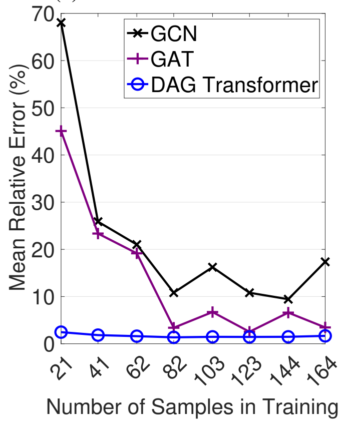
We evaluate the accuracy of GCN, GAT, and DAG Transformer for predicting the latency of stages of different benchmarks at various runtime configurations in two experimental platforms. For each scenario, we measure the mean relative errors (MREs) of each prediction model over different numbers of training samples from 10% to 80% of all the collected



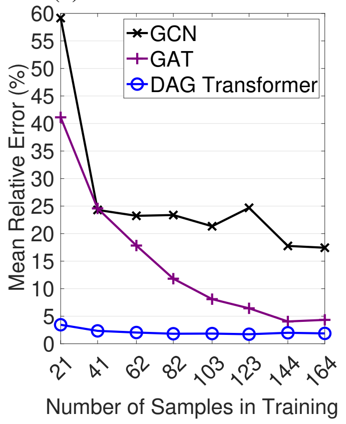
(a) GPT-3 on Platform 1



(b) GPT-3 on Platform 2

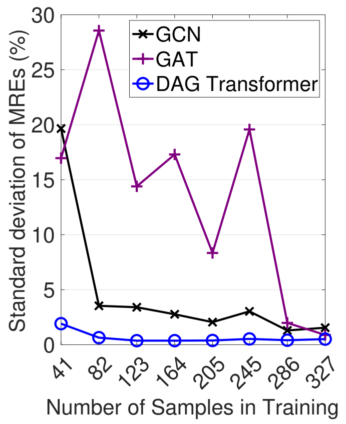


(c) MoE on Platform 1

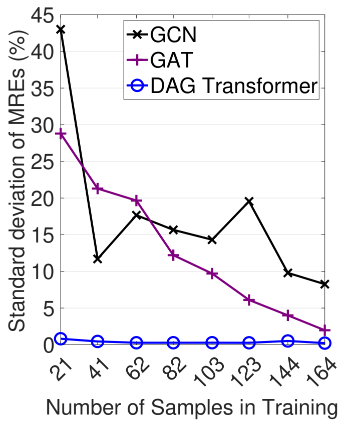


(d) MoE on Platform 2

Fig. 8: Average of MREs of different prediction models



(a) GPT-3 benchmark



(b) MoE benchmark

Fig. 9: Standard deviation of MREs of prediction models

samples an interval of 10% using separate 10% samples for validation and the remaining samples for testing and listed the measurements in Tbls V and VI. For a clearer view of the table data, we visualize the MREs of each prediction model on average over different pairs of mesh and configuration in Fig. 8 and plot their standard deviation over these scenarios on two platforms in Fig. 9.

Tbls V and VI show that DAG Transformer achieves the lowest MRE in 73.6% and 91.7% of scenarios for GPT-3 and MoE, respectively, and its MREs is consistently less than 3.7% when the number of training samples reaches 30% out of all the samples or beyond. This validates that DAG Transformer overperforms GCN and GAT in terms of prediction accuracy and also has a good reliability. From Fig. 9, it can be observed that the standard deviations of MREs for DAG Transformer do not exceed 2% and are much less than those of GCN and GAT, especially when the number of training samples reaches 30% out of all the samples or beyond. This illustrates that DAG Transformer is far more stable than GCN and GAT.

TABLE V: MRE (in %) at different runtime configurations in Platform 1 with NVIDIA A40 GPUs

(a) GPT-3 benchmark

# of Samples	Mesh 1			Mesh 2					
	Configuration 1			Configuration 1			Configuration 2		
	GCN	GAT	Tran	GCN	GAT	Tran	GCN	GAT	Tran
80%	**1.88**	4.56	2.33	4.55	4.04	**2.71**	1.79	3.22	**1.63**
70%	4.63	7.44	**2.89**	**2.63**	3.83	2.95	3.61	3.49	**2.62**
60%	5.81	66.74	**3.64**	4.51	6.43	**2.76**	3.89	3.60	**2.15**
50%	**2.57**	4.68	3.08	**1.80**	3.69	3.36	**1.82**	5.53	2.11
40%	**2.57**	3.93	3.47	**1.63**	35.40	2.91	**1.62**	57.03	2.44
30%	13.38	8.95	**3.01**	4.18	6.97	**3.22**	8.80	2.68	**2.31**
20%	10.37	5.05	**4.28**	3.97	9.45	**3.33**	2.37	3.88	**2.34**
10%	29.99	34.31	**3.60**	10.76	11.10	**4.28**	8.10	53.51	**6.97**

(b) MoE benchmark

# of Samples	Mesh 1			Mesh 2					
	Configuration 1			Configuration 1			Configuration 2		
	GCN	GAT	Tran	GCN	GAT	Tran	GCN	GAT	Tran
80%	11.94	4.85	**1.72**	17.70	4.07	**1.62**	22.43	**1.45**	1.67
70%	9.62	**1.20**	1.48	8.87	5.54	**1.40**	9.82	13.04	**1.53**
60%	6.35	**1.22**	1.65	13.93	2.10	**1.30**	12.14	4.29	**1.38**
50%	19.42	15.75	**1.54**	17.71	1.63	**1.54**	11.52	2.77	**1.31**
40%	3.82	1.61	**1.43**	8.43	4.14	**1.32**	20.23	4.42	**1.33**
30%	5.24	40.54	**1.66**	49.65	11.11	**1.42**	8.17	5.83	**1.73**
20%	50.72	22.54	**2.24**	12.72	17.90	**1.81**	14.15	29.61	**1.44**
10%	9.19	70.82	**1.90**	85.72	58.44	**3.36**	109.26	5.99	**2.12**

From Fig. 8, we also observe that the performance of GCN on the MoE benchmark is significantly worse than that of DAG Transformer. This is because the MoE stages typically involve larger graphs, where GCN fails to perform effectively. In contrast, the DAG Transformer consistently performs well across both benchmarks. We attribute this improvement to the DAG-based bias employed by the Transformer, which sets it apart from other general-purpose graph neural network (GNN) models. The DAG Transformer enables communication only between strongly connected nodes within the DAG using a single layer, facilitating more efficient learning on DAG data. In comparison, traditional GNN models, such as GCN and GAT, require stacking multiple layers to enable communication across a larger neighborhood, which is considerably less efficient.

B. *Use Case: To Reduce Optimization Cost in Parallelization Search for Distributed Training*

Our iteration latency predictor for distributed DL training can be applied to a variety of scenarios, including but not limited to high-throughput DL training service scheduling, elastic resource allocation for deadline-constrained DL training, and parallelization search for distributed DL training. In this subsection, we use parallelization search for distributed DL training as a case study to evaluate the practical effectiveness of our PredTOP. To reduce the optimization time of the well-known auto-parallelization tool, Alpa [42], which automates model-parallel training of large DL models by generating the optimal execution plan that unifies data, operator, and

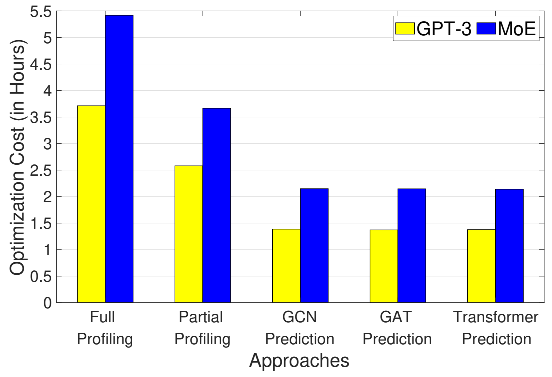
TABLE VI: MRE (in %) at different runtime configurations in Platform 2 with NVIDIA RTX A5500 GPUs

(a) GPT-3 benchmark

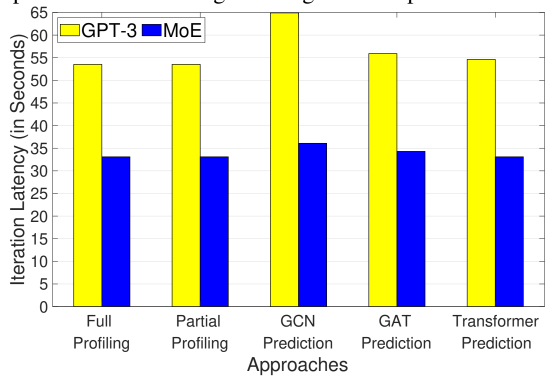
# of Samples	Mesh 1			Mesh 2						Mesh 3								
	Configuration 1			Configuration 1			Configuration 2			Configuration 1			Configuration 2			Configuration 3		
	GCN	GAT	Tran	GCN	GAT	Tran	GCN	GAT	Tran	GCN	GAT	Tran	GCN	GAT	Tran	GCN	GAT	Tran
80%	6.20	3.77	**3.37**	3.15	3.26	**2.47**	4.50	3.11	**1.86**	4.64	5.28	**2.18**	2.54	5.75	**2.42**	**1.42**	3.62	3.00
70%	4.50	3.93	**2.91**	5.94	3.37	**2.54**	4.33	5.22	**2.30**	1.63	9.19	**2.58**	2.05	2.91	**1.95**	**3.41**	4.74	3.47
60%	**2.01**	4.44	2.41	12.29	3.22	**2.65**	6.51	3.19	**1.85**	4.05	12.70	**2.34**	**2.06**	4.23	2.13	**2.36**	3.21	3.16
50%	7.23	3.97	**2.71**	6.89	30.55	**2.66**	**1.61**	3.97	2.35	2.92	3.92	**2.61**	3.45	11.17	**2.31**	**2.02**	3.20	3.05
40%	3.94	11.24	**2.97**	9.46	43.68	**2.88**	3.00	12.68	**2.36**	3.37	34.58	**2.43**	8.53	42.74	**2.31**	**2.03**	14.00	3.03
30%	4.24	15.39	**2.90**	7.37	8.92	**2.72**	7.25	9.34	**2.16**	**1.56**	52.77	2.68	10.26	3.49	**2.27**	4.85	12.66	**3.13**
20%	11.69	17.30	**3.72**	10.02	54.08	**3.42**	4.17	72.26	**2.22**	**2.38**	67.42	2.96	3.44	67.74	**2.47**	3.76	9.63	**3.19**
10%	9.86	10.49	**6.32**	63.03	34.95	**8.52**	47.15	43.41	**2.88**	7.21	46.73	**3.01**	15.43	17.27	**2.93**	**2.77**	4.40	4.73

(b) MoE benchmark

# of Samples	Mesh 1			Mesh 2						Mesh 3								
	Configuration 1			Configuration 1			Configuration 2			Configuration 1			Configuration 2			Configuration 3		
	GCN	GAT	Tran	GCN	GAT	Tran	GCN	GAT	Tran	GCN	GAT	Tran	GCN	GAT	Tran	GCN	GAT	Tran
80%	11.77	2.56	**2.30**	14.07	2.91	**1.58**	7.14	**1.85**	2.02	14.85	8.04	**1.68**	19.41	5.19	**1.87**	37.39	5.47	**1.81**
70%	10.95	5.81	**1.85**	11.88	**1.74**	1.88	14.19	**1.43**	1.64	10.82	3.24	**3.20**	16.87	1.81	**1.74**	41.87	10.26	**1.64**
60%	9.72	2.37	**2.29**	31.93	4.77	**1.68**	13.42	2.66	**1.62**	10.21	1.86	**1.58**	11.03	4.93	**1.65**	71.83	22.02	**1.55**
50%	8.78	4.44	**2.04**	11.15	2.45	**1.88**	16.66	2.29	**1.52**	26.23	1.95	**1.59**	8.13	5.00	**1.84**	57.07	32.48	**2.21**
40%	56.13	2.96	**2.08**	17.66	3.90	**1.93**	10.58	11.85	**1.78**	10.40	3.35	**1.51**	10.27	6.18	**1.63**	35.22	42.65	**1.97**
30%	17.99	18.60	**2.12**	3.71	8.40	**2.14**	52.28	7.97	**1.74**	30.65	4.18	**1.82**	7.69	3.25	**2.11**	27.10	64.62	**2.29**
20%	11.19	8.19	**2.31**	28.02	17.40	**2.16**	32.13	29.46	**2.22**	30.46	8.27	**2.34**	20.77	4.77	**1.84**	23.24	79.41	**3.15**
10%	17.15	69.37	**2.82**	88.08	31.06	**4.20**	19.53	27.68	**3.08**	18.18	20.00	**2.64**	125.79	8.08	**4.46**	86.18	90.56	**3.49**



(a) Optimization cost for generating the best parallelization plan



(b) Iteration latency of the optimized parallelization plan

Fig. 10: Parallelization plans optimized by Alpa integrated with PredTOP versus vanilla Alpa

pipeline parallelism, we integrated our PredTOP into Alpa. Originally, Alpa requires extensive profiling for parallelization plan generation. To reduce the total profiling time across all stages (referred to as "full profiling"), the vanilla Alpa employs a simple heuristic to selectively profile parts of the stages (referred to as "partial profiling"). Furthermore, we integrated Alpa with DAG Transformer-based PredTOP and its variants using GCN and GAT.

We measured the optimization time of vanilla Alpa using both full and partial profiling, as well as Alpa integrated with PredTOP and its two variants, on the GPT-3 and MoE benchmarks. The results are shown in Fig. 10a. From this figure, we observe that the optimization time of the DAG Transformer-based PredTOP is 46.6% lower for GPT-3 and 41.6% lower for MoE compared to vanilla Alpa with partial profiling, and is comparable to the optimization times of the GCN and GAT-based methods. This reduction is primarily due to the high cost of profiling, which involves enumerating the stages, running intra-operator optimization, compiling, transferring input data to the GPU, and performing profiling. Additionally, we noted that Alpa's inter-operator optimizer requires substantial memory for large models like MoE, limiting the ability to parallelize optimization processes of many stages simultaneously. This constraint further exacerbates the total optimization time, particularly in low-memory systems. In contrast, training a prediction model, which requires only minimal profiling, proves to be significantly more efficient.

To further evaluate the impact of PredTOP's prediction errors on the performance of the generated parallelization plan, we recorded the training iteration latency of the parallelization plan auto-optimized by these five versions for both benchmarks, with the results presented in Fig. 10b. We consider the iteration latency of the parallelization plan generated by vanilla Alpa with full or partial profiling as a baseline. Fig. 10b shows that the iteration latency of the parallelization plan optimized by Alpa integrated with the DAG Transformer-based PredTOP is only 2.1% higher for GPT-3 and identical for MoE compared to the baseline, due to the DAG Transformer's ability to accurately predict stage latency. However, the iteration latency of the parallelization plan optimized by Alpa integrated with PredTOP's two variants using GCN and GAT is 21.3% and 4.4% higher for GPT-3, and 9.0% and 3.6% higher for MoE, compared to the baseline, respectively. The poorer performance of the traditional GNN methods can be attributed to their lower prediction accuracy. During pipeline execution, a single stage is executed multiple times with different microbatches in each iteration, meaning that small errors in prediction can accumulate over the whole iteration, potentially becoming significant. Therefore, achieving high prediction accuracy for stage latency is crucial for ensuring accurate end-to-end latency.

IX. Related Work

There have been several efforts for latency prediction and performance modeling of DL training [8], [43]. We identify the existing prediction approaches as either *operator* or *workflow* based. Operator-based predictors focus on predicting the performance of individual DL operators, sometimes also referred to as a *kernel*. These approaches predict the performance of all the operators involved in a DL model training individually and combine the results to get the end-to-end latency. On the other hand *workflow*-based predictors first take a DL workflow which usually consists of several operators, usually organized as a computational graph, and try to predict the performance of the DL workflow holistically.

Both operator and workflow based techniques usually rely on either an analytical or data-driven model to predict the runtime information of DL models, which has been inadequate for predicting runtime information for training with hybrid parallelism. Hierarchical modeling combining both data-driven and analytical models, may be utilized to properly represent the complexity involved DL training with hybrid parallelism, such as the proposed PredTOP.

A. Operator-based modeling

Black-box modeling: ETS [39] proposed a LSTM-based prediction technique on slices of execution trace to predict the execution time of DL operators as a sequence. Habitat [40] also employed a MLP model to predict the runtime information for some operators. While these techniques have seen success in single device training, they do not consider multi-device parallel training.

White-box modeling: Paleo [25] decomposed DL model layers into their computation and communication components and proposed an analytical model for performance prediction of DL model. However, this approach focused primarily on CNNs and used a simple model. It relied on metrics such as FLOPS, which is shown to be unreliable in modern DNN models such as transformers [4]. Habitat [40] proposed a scaling method utilizing analytical techniques to make runtime predictions on one device based on its performance on another device by utilizing various device properties.

Operator-based techniques can accurately model the complexity of individual operator execution, but they lack the holistic view of the DL model and fail to predict accurate end-to-end latency.

B. Workflow-based modeling

Black-box modeling: DNNperf [7] proposed a GNN with an attention-based node-edge encoder to predict the runtime performance of a deep learning model. Similarly, Driple [38] proposed a model for predicting resource consumption on distributed training. In order to make the approach applicable to different runtime environments, they also proposed a transfer learning technique. While both these methods showed good results in specific scenarios, they rely on very large general models for performance prediction, which requires a large amount of training time and data. In addition, they also fail to encompass various parallel training scenarios. PredictDLL [2] suggested a Graph HyperNetwork based technique to extract DL model feature as an embedding vector and a regression model to predict the performance. We found this work to be targeted for cloud-based training and not applicable on large-scale parallel training.

White-box modeling: Cheng et al. [4] performed a thorough study to evaluate the computation and communication patterns in large-scale distributed transformer training. This study has presented some invaluable insights for transformer training, but not general DL models. Similarly, FasterMoE [9] built computation and communication models separately and designed a distributed deep learning roofline model to combine the two. While this was used to optimize MoE model training, it fails to provide a general purpose latency prediction system for distributed DL training.

Workflow-based modeling generally relies on large data-driven models, which try to predict the runtime information, but it fails to be generalized to different training scenarios and environments. Fortunately, these techniques may be integrated with traditional workflow modeling methods [13], [28]–[33].

X. Conclusion

The use of pure black-box or white-box modeling approaches has been considered infeasible for performance prediction of distributed DL training with hierarchical parallelism due to large search spaces of hybrid parallelism, high measurement cost of large-scale training, and complex interactions among operator partitions or operators. PredTOP can accurately and efficiently predict the iteration latency of distributed DL training with pipeline, model, and tensor parallelism by employing DAG Transformer technique into black-box modeling of stage latency and leveraging the low cost of white-box modeling for a pipeline consisting of stages. Experimental results show that PredTOP can help an automatic parallelization plan generation system save the optimization cost by up to 46.6% at a cost of no more than 2.1% performance degradation of the optimized parallelization plan.

Acknowledgment

This research is sponsored by U.S. National Science Foundation under Grant No. OAC-2306184 and Grant No. OAC-2443633 with the University of North Texas.

REFERENCES

[1] JAX. https://docs.jax.dev/en/latest/. [Accessed 13-02-2025].

[2] K. Assogba, E. Lima, M. M. Rafique, and M. Kwon. PredictDDL: Reusable workload performance prediction for distributed deep learning. In Proc. of IEEE International Conference on Cluster Computing (Cluster), pages 13–24, Santa Fe, NM, USA, Oct-Nov 2023.

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners, 2020. https://splab.sdu.edu.cn/GPT3.pdf.

[4] S. Cheng, J.-L. Lin, M. Emani, S. Raskar, S. Foreman, Z. Xie, V. Vishwanath, and M. T. Kandemir. Thorough characterization and analysis of large transformer model training at-scale. In Proc. of ACM on Measurement and Analysis of Computing Systems (SIGMETRICS), volume 8, no. 1, pages article 8:1–25, Venice, Italy, Jun 2024.

[5] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, L. Diao, X. Liu, and W. Lin. DAPPLE: A pipelined data parallel approach for training large models. In Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 431–445, Virtual, Feb 2021.

[6] R. Frostig, M. J. Johnson, and C. Leary. Compiling machine learning programs via high-level tracing. In SysML Conference (SysML), Stanford, CA, USA, Feb 2018. https://mlsys.org/Conferences/doc/2018/146.pdf.

[7] Y. Gao, X. Gu, H. Zhang, H. Lin, and M. Yang. Runtime performance prediction for deep learning models with graph neural network. In Proc. of IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pages 368–380, Melbourne, Australia, May 2023.

[8] Y. Gao, Y. Liu, H. Zhang, Z. Li, Y. Zhu, H. Lin, and M. Yang. Estimating gpu memory consumption of deep learning models. In Proc. of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 1342–1352, Sacramento, CA, USA, Nov 2020.

[9] J. He, J. Zhai, T. Antunes, H. Wang, F. Luo, S. Shi, and Q. Li. FasterMoE: Modeling and optimizing training of large-scale dynamic pre-trained models. In Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 120–134, Virtual, Apr 2022.

[10] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. In Proc. of Conf. on Neural Information Processing Systems (NeurIPS), volume 32, Vancouver, BC, Canada, Dec 2019.

[11] S. Hwang, E. Lee, H. Oh, and Y. Yi. FASOP: Fast yet accurate automated search for optimal parallelization of transformers on heterogeneous GPU clusters. In Proc. of Intl. Symposium on High-Performance Parallel and Distributed Computing (HPDC), pages 253–266, Pisa, Italy, Jun 2024.

[12] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In Proc. of Intl. Conf. on Learning Representations (ICLR), pages 1–14, Toulon, France, Apr 2017.

[13] T. Kundu and T. Shu. HIOS: Hierarchical inter-operator scheduler for real-time inference of DAG-structured deep learning models on multiple GPUs. In Proc. of the 25th IEEE International Conference on Cluster Computing (Cluster), pages 95–106, Santa Fe, NM, USA, Nov 2023.

[14] S. Li and T. Hoefler. Chimera: Efficiently training large-scale neural networks with bidirectional pipelines. In Proc. of ACM/IEEE International Conference for High-Performance Computing, Networking, Storage and Analysis (SC), St. Louis, MO, USA, Nov 2021.

[15] Y. Li, J. Baik, M. M. Rahman, I. Anagnostopoulos, R. Li, and T. Shu. Pareto optimization of CNN models via hardware-aware neural architecture search for drainage crossing classification on resource-limited devices. In Proc. of Workshops of ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC-W), pages 1767–1775, Denver, CO, USA, Nov 2023.

[16] Z. Lin, Y. Miao, Q. Zhang, F. Yang, Y. Zhu, C. Li, S. Maleki, X. Cao, N. Shang, Y. Yang, W. Xu, M. Yang, L. Zhang, and L. Zhou. nnScaler: Constraint-guided parallelization plan generation for deep learning training. In Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 347–363, Santa Clara, CA, USA, Jul 2024.

[17] Z. Liu, S. Cheng, H. Zhou, and Y. You. Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency. In Proc. of ACM/IEEE International Conference for High-Performance Computing, Networking, Storage and Analysis (SC), Denver, CO, USA, Nov 2023.

[18] Y. Luo, V. Thost, and L. Shi. Transformers over directed acyclic graphs. In Proc. of Conf. on Neural Information Processing Systems (NeurIPS), volume 36, pages 47764–47782, New Orleans, LA, USA, Dec 2023.

[19] X. Miao, Y. Wang, Y. Jiang, C. Shi, X. Nie, H. Zhang, and B. Cui. Galvatron: Efficient transformer training over multiple GPUs using automatic parallelism. In Proc. of the VLDB Endowment (VLDB), volume 16, no. 3, pages 470–479, Vancouver, BC, Canada, Aug-Sep 2023.

[20] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In Proc. of ACM Symposium on Operating Systems Principles (SOSP), pages 1–15, Huntsville, ON, Canada, Oct 2019.

[21] A. Nazeri, D. W. Godwin, A. M. Panteleaki, I. Anagnostopoulos, M. I. Edidem, R. Li, and T. Shu. Exploration of tpu architectures for the optimized transformer in drainage crossing detection. In Proc. of IEEE Intl. Conf. on Big Data (BigData): 5th Intl. Wksh. on Big Data and AI Tools, Methods, and Use Cases for Innovative Scientific Discovery (BTSD), pages 4178–4187, Washington DC, USA, Dec 2024.

[22] D. Pandey, J. Ghebremichael, Z. Qi, and T. Shu. A comparative survey: Reusing small pre-trained models for efficient large model training. In Proc. of Workshops of IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC-W), pages 56–63, Atlanta, GA, USA, Nov 2024.

[23] D. Pandey and T. Shu. AM-DGCNN: Leveraging graph attention networks and edge attributes for link classification in knowledge graphs. In Proc. of Workshops of IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC-W), pages 1037–1045, Atlanta, GA, USA, Nov 2024.

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In Proc. of Conf. on Neural Information Processing Systems (NeurIPS), volume 32, Vancouver, BC, Canada, Dec 2019.

[25] H. Qi, E. R. Sparks, and A. Talwalkar. Paleo: A performance model for deep neural networks. In Proc. of the Intl. Conf. on Learning Representations (ICLR), Toulon, France, Apr 2017.

[26] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. ZeRO: Memory optimizations toward training trillion parameter models. In Proc. of ACM/IEEE International Conference for High-Performance Computing, Networking, Storage and Analysis (SC), vitrual event, Nov 2020.

[27] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism, 2019. https://arxiv.org/pdf/1909.08053.

[28] T. Shu, Y. Guo, J. Wozniak, X. Ding, I. Foster, and T. Kurc. Bootstrapping in-situ workflow auto-tuning via combining performance models of component applications. In Proc. of ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), pages 1–15, St. Louis, MO, USA, Nov 2021.

[29] T. Shu, Y. Guo, J. Wozniak, X. Ding, I. Foster, and T. Kurc. POSTER: In-situ workflow auto-tuning through combining component models. In Proc. of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 467–468, Seoul, South Korea, Feb-Mar 2021.

[30] T. Shu and C. Q. Wu. Energy-efficient mapping of big data workflows under deadline constraints. In Proc. of the 11th Workshop on Workflows in Support of Large-Scale Science (WORKS) in conjunction with ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pages 34–43, Salt Lake City, UT, USA, Nov 2016. http://ceur-ws.org/Vol-1800/paper5.pdf.

[31] T. Shu and C. Q. Wu. Energy-efficient dynamic scheduling of deadline-constrained MapReduce workflows. In Proc. of the 13th IEEE International Conference on eScience (e-Science), pages 393–402, Auckland, New Zealand, Oct 2017.

[32] T. Shu and C. Q. Wu. Performance optimization of Hadoop workflows in public clouds through adaptive task partitioning. In Proc. of IEEE International Conference on Computer Communications (INFOCOM), pages 2349–2357, Atlanta, GA, USA, May 2017.

[33] T. Shu and C. Q. Wu. Energy-efficient mapping of large-scale workflows under deadline constraints in big data computing systems. 110:515–530, 2020.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Łukasz Kaiser, and I. Polosukhin. Attention is all you need. In Proc. of Conf. on Neural Information Processing Systems (NeurIPS), volume 30, pages 5998–6008, Long Beach, CA, USA, Dec 2017.

[35] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In Proc. of the Intl. Conf. on Learning Representations (ICLR), Vancouver, BC, Canada, May 2018.

[36] C. Wang, D. Sun, and Y. Bai. PiPAD: Pipelined and parallel dynamic GNN training on GPUs. In Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 405–418, Montreal, Canada, Feb 2023.

[37] Y. Xia, Z. Zhang, H. Wang, D. Yang, X. Zhou, and D. Cheng. Redundancy-free high-performance dynamic GNN training with hierarchical pipeline parallelism. In Proc. of ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC), pages 17–30, Orlando, FL, USA, Aug 2023.

[38] G. Yang, C. Shin, J. Lee, Y. Yoo, and C. Yoo. Prediction of the resource consumption of distributed deep learning systems. In Proc. of ACM on Measurement and Analysis of Computing Systems (SIGMETRICS), volume 6, no. 2, pages article 29:1–25, Mumbai, India, Jun 2022.

[39] Z. Yang, H. Guo, H. Wu, Y. Wu, H. Zhong, W. Zhang, C. Zhou, and Y. Liu. ETS: Deep learning training iteration time prediction based on execution trace sliding window. In Proc. of Intl. Symposium on High-Performance Parallel and Distributed Computing (HPDC), pages 56–68, Pisa, Italy, Jun 2024.

[40] G. X. Yu, Y. Gao, P. Golikov, and G. Pekhimenko. Habitat: A runtime-based computational performance predictor for deep neural network training. In Proc. of USENIX Annual Technical Conference (ATC), pages 503–521, Virtual, Jun 2021.

[41] Y. Zhang, D. Pandey, D. Wu, T. Kundu, R. Li, and T. Shu. Accuracy-constrained efficiency optimization and GPU profiling of CNN inference for detecting drainage crossing locations. In Proc. of Workshops of ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC-W), pages 1780–1788, Denver, CO, USA, Nov 2023.

[42] L. Zheng, Z. Li, H. Z. Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 559–578, Carlsbad, CA, USA, Jul 2022.

[43] H. Zhu, A. Phanishayee, and G. Pekhimenko. Daydream: Accurately estimating the efficacy of optimizations for DNN training. In Proc. of USENIX Annual Technical Conference (ATC), pages 337–352, Virtual, Jul 2020.