36th European Modeling & Simulation Symposium
21st International Multidisciplinary Modeling & Simulation Multiconference

2724-0029 © 2024 The Authors.
doi: xx.xxxx/xxxx

Tianshu: Towards Accurate Measuring, Modeling and Simulation of Deep Neural Networks

Hongming Huang[1,*], Hong Xu[2] and Nan Guan[1]

[1]Department of Computer Science, City University of Hong Kong, Hong Kong SAR, China
[2]Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong SAR, China

[*]Corresponding author. Email address: honhuang7-c@my.cityu.edu.hk

Abstract

To help train DNN models more efficiently, researchers have developed a series of new computation devices and parallel training strategies. Choosing which device and strategy to use among those vast candidates is challenging for developers. Simulation is an effective solution to this question since it does not need to deploy models with different settings practically. However, most existing works still require users to measure models on the target hardware first, which cannot help users choose the proper hardware in advance. This paper presents Tianshu, which adopts the *ahead-of-time* idea to solve this problem. Tianshu first designs an accurate measuring tool to measure the operation's time-use and store the result in a database, which can be shared among the community. For uncovered cases, Tianshu designs an automatic mechanism to train neural network models that can estimate their execution time. Therefore, Tianshu can support simulation without a target hardware. Finally, Tianshu leverages a discrete event simulator to simulate the DNN model's execution. Evaluation on Azure clusters with 8x V100 GPUs shows that Tianshu achieves 93% average accuracy for six well-known DNN models. Tianshu's operation estimator also achieves 96% average accuracy on 9 typical operations and outperforms existing works.

Keywords: Neural Networks; Artificial Intelligence; Discrete Event Simulation

1. Introduction

Recently, Deep Neural Networks (DNNs) have achieved remarkable success in many machine learning tasks, including neural language processing, image recognition, object detection, auto-pilot, etc (Achiam et al., 2023; Devlin et al., 2019; He et al., 2016; Redmon et al., 2016). This success has made DNN more and more popular for academic research and industrial applications. However, DNN models usually have intensive computation overhead, which is not affordable for traditional CPU-based machines. Therefore, the acceleration of DNN training has attracted more and more researchers' interest, including new computation hardware like GPU and TPU (Jouppi et al., 2017), communication hardware like PCI-e and NVLink (Foley and Danskin, 2017), and training strategies like data parallel and pipeline parallel (Osawa et al., 2023).

Though these efforts have provided us with a vast number of powerful tools, nowadays, users have another question: How can we choose the proper hardware and strategy to deploy the DNN model or build the machine learning cluster? More specifically, the question can be divided into two sides:

- **Deciding the target hardware to buy or rent.** Current hardware providers like Nvidia produce many different GPUs with various computation capabilities and prices(e.g., T4, P100, V100, and A100). Besides the computation devices, there are also different communication hardware (e.g., PCIe, CXL, and NVLink). These different hardware offer different trade-offs between performance and financial cost.
- **Deciding the training strategy for models.** DNN model size is undergoing a continuous scaling-up (e.g., BERT, DALL-E, GPT-3 and GPT-4 (Devlin et al., 2019; Ramesh

et al., 2021; Achiam et al., 2023)). Therefore, paralleled training has become indispensable for DNN training. Researchers have proposed many strategies, such as data parallelism, model parallelism, and pipeline parallelism (Shoeybi et al., 2019; Osawa et al., 2023). Choosing proper training strategies for these colossal models is also essential to improve the DNN model's performance.

These questions are hard to answer because the best choice may not exist. Users need to evaluate whether spending more budget on more powerful hardware can bring a worthwhile performance gain. However, the performance gain depends not only on the hardware but also on the specific model. According to our experiment, the training throughput of the light-weight LSTM model on the V100 GPU is just the same as the P100 GPU, while for the VGG16 model, the V100 GPU can promote the throughput by 30% over the P100 GPU. In this case, selecting the most powerful GPU not always improves performance while introducing extra financial costs.

An intuitive solution for this question is directly measuring the target DNN model's performance on the real device. The major drawback of this approach is that users have to get access to the target device and deploy their model. Then, users also need to try different training strategies. Even though the user can rent some devices from cloud service providers like Azure or Amazon, the long training process for large models will make such repetitive measuring with different devices and strategies inefficient and expensive.

Some related works (Geoffrey et al., 2021; Jia et al., 2019; Narayanan et al., 2019) propose that simulation can help users understand their model's performance under different training strategies without deploying them on the cluster. However, most of them still require users to measure the DNN model on one target device. Therefore, they cannot help users decide which hardware to purchase or rent. Another common problem of existing solutions is that they all rely on the built-in profiler of machine learning frameworks (like TensorFlow) or the GPU profiler from Nvidia to do the measurement. However, our experiment shows that these measuring tools can produce over 50% of measuring errors for DNN models with tiny operations like LSTM.

This paper focuses on designing an accurate simulator for DNN models to answer the above two questions. Our key idea is that the measurement can be done ahead-of-time (AOT). The idea is based on two insights. **First**, most DNN models are made up of common basic units, called *operations*. **Second**, parameters that determine an operation's execution time can be acquired from the model's dataflow graph without the target hardware. These insights imply that we can measure operations offline and store the result (including operations parameters and execution time) in a database for future usage and even share the database among the community. Therefore, users can simulate their DNN model's performance using existing measuring data from other users.

To achieve our goal, we still need to overcome two main challenges. The first is how to accurately measure the execution time of tiny operations, which is the basis of simulation. The second is that the database cannot cover all possible operation parameters. How can we estimate the operator's execution time with unseen parameters?

In this paper, we propose a novel DNN simulation framework named Tianshu (Tianshu means the star of the celestial pivot, which is the Chinese name for the alpha star of the constellation of Ursa Major. It is used to guide the north direction in ancient China). Tianshu consists of three major components. The first module is an accurate **measuring tool**. This tool measures a target operation by measuring the execution time of dataflow graphs with N and M copies of operations and then calculating the time for a single operation. The second module is an **estimator** for the operation's execution time. It leverages feature engineering to infer the operation's theoretical time complexity and automatically trains a neural network model to predict its execution time. The last module is a **discrete event simulator** to simulate the training process of the DNN model.

We implement Tianshu with the popular framework TensorFlow and evaluate it using six well-known DNN models on two servers with 8x V100 GPUs on the Azure cloud. The measuring tool of Tianshu achieves 95% average accuracy with a maximum error of 8%, while traditional measuring tools have a maximum error of 86%(TensorFlow profiler) and 49%(Nvprof). The estimation model of Tianshu achieves 96% average prediction accuracy on 9 typical operations, and the feature engineering of the estimation model effectively reduces the prediction error by 74%. Finally, we evaluate the simulation of end-to-end DNN training on 1, 2, 4, and 8 GPUs. Tianshu achieves 93% average accuracy for all six models and over 95% accuracy for CNN models.

To summarize, this paper makes the following contributions:

- We build a novel DNN model performance simulator Tianshu based on our key idea: ahead-of-time measurement. The experiment proves that Tianshu can achieve high simulation accuracy for different kinds of DNN models.
- We design a new operation measuring tool for TensorFlow, which reduces the measuring error for minor operations by 90% compared to existing tools.
- We design an estimation model with an original feature engineering mechanism to predict the execution time of operations with unseen parameters accurately.

2. Research Background

We start by introducing the background of this research topic and then develop our key idea.

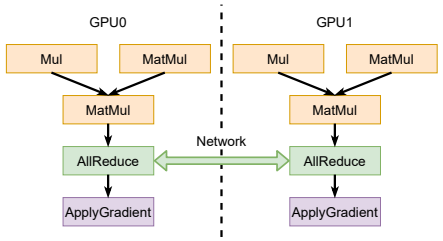

Figure 1. Dataflow graph example for part of the gradient calculation for one layer of AlexNet (Krizhevsky et al., 2012). `MatMul` calculates the gradients using output of two previous operations. The `AllReduce` operation synchronizes gradients on all devices, and `ApplyGradient` applies the gradients to model weights.

2.1. Background and Related Work

Deep Neural Network(DNN) Models. At the high level, all machine learning systems follow a similar workflow: The user first provides a DNN model with different layers (e.g., convolution layer) to be trained, often using the system's Python API (e.g., PyTorch, TensorFlow). The system then transforms the model into a *dataflow graph* (Zaccone, 2016; Abadi et al., 2016) , where each node represents an operation (e.g., `Conv2D` operation for 2-dimension convolution, or `AllReduce` operation for synchronizing gradients among different GPUs), and each edge represents a dependency between two operations. According to this *dataflow graph* with Figure 1 as an example, the operations are sent to devices such as GPU and network for execution.

A computation operation like `Conv2D` is eventually implemented by low-level libraries like cuDNN to one or more specific functions (*GPU kernels*) for best performance. The information of operations can be directly extracted from the dataflow graph. However, the GPU kernel's information can only be recorded by specific runtime profiling tools, and it is tough to analyze which operation a GPU kernel belongs to. Therefore, most existing works are conducted at the operation level.

Existing works. ML system simulation has received limited attention in the community until very recently. *Habitat* (Geoffrey et al., 2021) only considers single operation's performance. It tries to train a DNN model for each operation to predict its execution time on a specific GPU with the measuring result on another type of GPU. The problem is that it requires a large amount of training data on all target devices because it needs to train N^2 models for N hardware types. *Daydream* (Zhu et al., 2020) focuses on modeling various DNN optimizations with a set of graph-transformation primitives and estimating their effect. E.g., the user can manually adjust the execution time of a specific operation and see the change of the entire model's training time. Besides, *FlexFlow* and *PipeDream* (Jia et al., 2019; Narayanan et al., 2019) incorporate a simulator as an auxiliary tool to compare different strategies. They first measure the execution time for all operations in the model and then leverage a simple event-driven simulator to predict the model's training time.

The key problem with existing works is that they rely on online measuring, which means they always conduct the measuring before the simulation. This online measuring is very expensive and does not scale. Users have to purchase or rent the hardware and build the platform first, which means considerable financial and time costs. Especially for users who are looking to purchase cost-efficient GPUs for their purposes, they would ideally want to know the performance of their target DNN models before spending money to buy GPUs. Then, whenever users want to modify the model and the input size or even restart the simulation with a different parameter, they need to repeat the measuring.

Another problem is the measuring tool itself. Currently, there are two main measuring approaches, both of which have drawbacks. The first one is the *built-in profiler* of the machine learning framework, e.g., the TensorFlow profiler. The problem with the TensorFlow profiler is that it relies on intrusive probes to record execution information of operations. Consequently, it introduces overhead to the execution time and generates significant measurement errors for tiny operations. The second approach is the *hardware profiler* like Nvidia's Nsight or Nvprof, which directly gathers information from GPUs. The drawback of these profilers is that they only record the execution time of GPU computation functions (GPU kernels) while omitting the time used for task scheduling and data transmission. According to our experiment of measuring the LSTM model on a V100 GPU, the total execution time of all operations measured by the TensorFlow profiler is 86% larger than the real execution time, and the Nvprof's result is 30% smaller than the real time.

2.2. Our Idea: Ahead-of-Time Measuring

After investigating many different DNN models, we found two important observations and proposed our idea.

Observation 1: Common operations for different models. Though a DNN model may contain thousands of operations, most of them are usually made up of a small set of common operations. For example, different convolution network models will share similar convolution, fully connected, and pooling operations, though they have distinct input shapes or parameters in different models. These common operations make up the majority of the model's computation workload and iteration time.

Observation 2: Available operation information. Even if the user does not have a GPU, the mainstream machine learning framework like TensorFlow can still generate the dataflow graph of the DNN model. This dataflow graph contains valuable information, including each operation's input data shape and parameters. This information defines an operation's task, and two operations with the same input data shape and parameters naturally have the same execution time.

Key idea: Based on the two observations, we propose our idea of conducting measurements *ahead-of-time*. We can

measure different operations in various settings just once and save the results in a database. Then, all future simulations can reuse data in the database without repetitive measuring. Another essential benefit of ahead-of-time measuring is that users can share data among the community. Leveraging other users' measuring data allows users to do simulations without accessing expensive GPUs and proprietary hardware platforms. Such new-generation simulators can enable researchers to predict the performance of different hardware and training strategies, make informed decisions in optimizing their systems, and plan their private clusters in advance.

Challenges: To realize our goal, we need to overcome two challenges. **First**, how could we accurately measure those small operations? Existing measuring methods still have huge errors when handling them. Without accurate measuring, there will be no precise simulation. **Second**, the database cannot cover all possible values of parameters. Therefore, we have to build a model to estimate the operation's execution time with uncovered parameters in the database.

3. Design of Tianshu

After discussing our key idea and challenges, this section presents Tianshu's solution to them. We start by introducing Tianshu's measuring method in Sec 3.1, and then discuss how to model and estimate operation's execution time in Sec 3.2. Finally, we present the discrete event simulation for the DNN model in Sec 3.3.

3.1. Accurate Measurement of Operations

The primary difficulty of measuring lies in small operations. Directly measuring the operation will introduce negligible error because of TensorFlow's mechanism: TensorFlow's execution is invoked by a *Session* with an assigned list of target operations in the dataflow graph. The Session will execute all required operations and return results of the target operations. The launch of the Session has a millisecond-level overhead, which harms the accuracy of direct measurement, especially for small operations that only have a microsecond-level execution time. Furthermore, users cannot add probes to record the execution of operations without directly modifying the source code of TensorFlow, which will strongly bind the measuring tool to a specific version of TensorFlow.

Tianshu's solution leverages a common statistic approach to eliminate such measuring errors. It first automatically analyzes the target operation's structure and builds two measurement dataflow graphs with N and M copies of the target operation, and adding some constant tensors with random values to serve as the input data (E.g., for `Add`, there are two input tensors). Figure 2 shows the structure of measurement dataflow graphs. A notable thing is that we add a dependency between the *i-th* and *(i+1)-th* operations to ensure that all operations are executed sequentially without parallelism. Then, the execution time of the whole dataflow graph should contain the constant overhead of launching the session and the time use of N or M target operations. After we measured the execution time of two graphs, we can infer the time use of one target operation by:

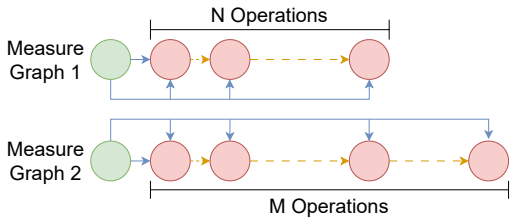

Figure 2. Example of how Tianshu construct measurement dataflow graphs. Green circles are constant input tensors and red circles are copies of target operations. Blue arrows are dataflow, while orange arrows with dashed lines are execution dependencies.

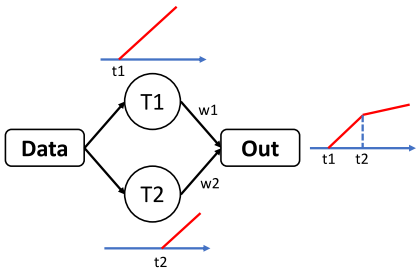

Figure 3. Example of how neural networks with ReLu activation approximates a non-linear function. A neuron generates a linear function in a right open interval $[t_1, \infty)$ (or a left open interval with negative weights). The neuron in the next layer takes two such inputs with different interval terminals and generates a piece-wise linear function.

$$T_{op} = \frac{T_M - T_N}{M - N} \tag{1}$$

This measuring method has two advantages: First, it does not need to modify the framework's code and supports all framework versions. Second, it eliminates systematic errors and does not introduce any other errors to the measuring or omit execution time. This accurate measuring tool will act as the foundation of Tianshu.

3.2. Modeling and Estimation of Operations

Since Tianshu's database cannot cover all possible inputs and parameters, we need to build a model using existing measurement data to estimate the operation's execution time with inputs that are not recorded. However, manually building estimation models for each operation is impractical due to the number of operations. E.g., TensorFlow 1.15 contains over 1300 kinds of operations.

Tianshu considers this estimation task as a non-linear regression problem: Given a series of input pa-

rameters of an operation (including the input tensors' shapes and other parameters), predict the execution time. Tianshu adopts a neural network model Multilayer-Perceptrons(MLP), which has been used for a long time to handle such problems (Rynkiewicz, 2012; Pal and Mitra, 1992). However, the MLP model has its limitations. Figure 3 shows how MLP fits non-linear functions by a piece-wise linear fitting approach. For complex non-linear target functions, MLP models need to split the definition domain into lots of segments, which requires numerous neurons, model parameters, and exponential growth of training data.

Tianshu's idea for this challenge is that the theoretical time complexity could facilitate MLP models in learning the non-linear target function. In Sec 3.2.1, we first investigate the relationship between features and the execution time. Then, Tianshu exploits this relationship through feature engineering, which generates a new feature called *principal Component (PC)* that represents the theoretical time complexity in Sec 3.2.2. Finally, Tianshu leverages an automatic training mechanism to search for the proper model structure and train the MLP model in Sec 3.2.3.

3.2.1. *Investigation of Parameters and Theoretical Time Complexity*

As a specific algorithm, every operation has its theoretical time complexity, which is a function of input parameters. For example, the matrix multiply operation's time complexity is $n * m * k$, while the 2-dimension convolution operation `Conv2D`'s time complexity is:

$$T_{Conv2D} = \frac{n * h * w * c * filter_x * filter_y}{stride_h * stride_w} \quad (2)$$

The n, h, w, c are the shape of input data, $filter_{x,y}$ is the size of the convolution kernel, and $stride_{h,w}$ are the step length of the sliding window. With the precise guidance of theoretical time complexity, MLP models can learn to predict the operation's execution time more effectively.

Due to the large number of different kinds of operations, it is impractical to analyze their theoretical time complexity manually. In order to explore an automatic approach to finding the theoretical time complexity, we first investigate the relationship between parameters and the execution time of hundreds of kinds of operations in six well-known models (see Table 1 in Sec 4.1), and classify these relationships into four distinct types:

- *Linear*: The execution time changes linearly with the parameter. For example, Figure 4 depicts the change of execution time with the row dimension k of the matrix multiply operation.
- *Inversely-proportional*: The execution time is inversely-proportional to the parameter. Figure 5 presents how *strides_h* alters the execution time.
- *Null*: The parameter has no impact on the execution time, like the *value* parameter of `Const` operation (assigns the value of a constant tensor).
- Mode selection: The parameter's different value will totally changes the implementation of the operation, like the *data_type*. Different data types (integer or real number) will lead to different GPU functions.

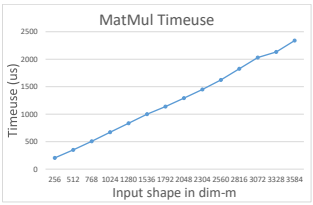

Figure 4. Impact of matrix a's row dimension m to the execution time of `MatMul`. The n and m (row/column dimension of matrix a/b) are fixed at 2048.

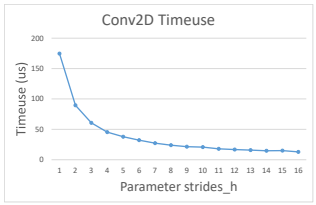

Figure 5. Impact of parameter *strides_h* to the execution time of `Conv2D`. The input shape is [64, 512, 512, 3] and the filter size is. [3, 3, 32].

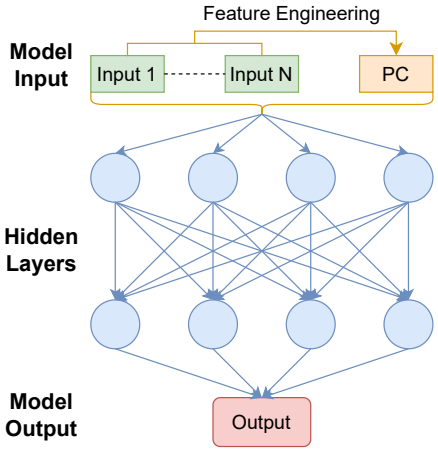

Figure 6. Overview of Tianshu's operation estimation model. It first generate the principal component (PC) as an extra input feature, and then feed it into the estimation model.

This finding greatly reduces the number of possible function structures of theoretical time complexity. Based on this finding, we design an automatic feature engineering mechanism for Tianshu to construct the principal component using information of parameters.

3.2.2. *Feature Engineering*

Though the investigation in Sec 3.2.1 has greatly reduced the number of possible time complexity function's formation, there are still 4^N possible functions for an operation with N parameters. Tianshu conducts the feature engineering by first pruning the number of possible function structures according to some empirical rules. Then, it automatically tries all possible functions and selects the best function to build the principal component. Fig 6 shows how the principal component acts as an extra input to help train the prediction model.

Step 1: Pruning. In the first step, Tianshu reduces the number of possible time complexity functions using the

following rules, which are summarized from the investigation:

- If a parameter only has no more than 5 possible values, we regard it as a mode selection parameter.
- If a parameter is a character string, it can only be a null or mode selection parameter.
- If a parameter is a dimension of the input tensors' shape, it can only be a linear or null parameter.

The pruning step greatly reduces the number of possible time complexity functions. For example, the matrix multiply operation has 5 parameters: 4 are the shape of two input matrices, and the other is the data type. Without pruning, Tianshu needs to examine $4^5 = 1024$ possible time complexity functions. After pruning, the data type is determined as a mode selection parameter, and the input tensor shape parameter can only be linear or null parameter. Consequently, there are only $2^4 = 16$ possible time complexity functions for the second step to examine.

Step 2: Automatic searching of principal component. In this step, Tianshu examines all possible time complexity functions individually. Each time, Tianshu picks one function and calculates the principal component with it. Then, Tianshu adds the function value as an extra input parameter to a default MLP model with 6 fully connected layers, and each layer contains 6 neurons (like Figure 6 shows). Tianshu trains the model with default hyper-parameter settings and records the final prediction accuracy. The candidate function that yields the best accuracy is selected as the principal component of the model.

3.2.3. *Building and Training of Estimation Model*

After the principal component is determined for the target operation, Tianshu builds and trains the prediction model in two steps. It first searches for the best model architecture and then conducts the fine-tune training.

Step 1: Architecture searching. Tianshu leverages the AutoML techniques to search the model architecture in a given range automatically. We set the range of the number of layers to [3,20], and the number of neurons per layer to [3,40]. Tianshu will train all these models using the same default hyper-parameters. Then, the top-5 models with the highest prediction accuracy on the test set will be kept in the fine-tune training step.

Step 2: Fine-tune training. In this stage, Tianshu trains all models with different hyper-parameters, including the learning rate, batch size and different initialization values. Eventually, Tianshu selects the best model as the final estimation model.

For each value of the *mode selection* parameters, Tianshu trains an independent estimation model. These models use the same principal component since they serve the same operation algorithm. However, they require independent architecture searching and deep training procedures to capture different implementations of different modes (like different GPU functions for different data types).

Tianshu's estimation model training technique has three main advantages: 1) The entire process are automatically performed offline without human's interaction. 2) Users can share trained models like the database among the community, and other users can directly use them without training. 3) The fine-tune training phase can be repeated to keep improving the model whenever the database collects more data.

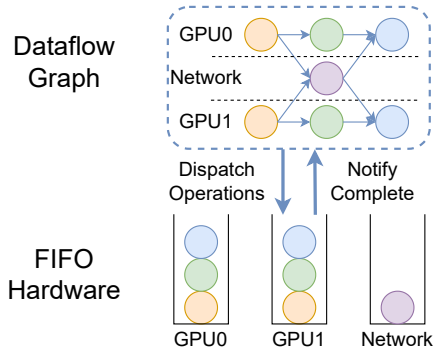

Figure 7. Discrete event simulator of Tianshu. Green circles are virtual computation operations generated for the purple `AllReduce` operation to simulate its interaction with computation tasks.

3.3. Discrete Event Simulation of DNN Models

After all computation operations' execution time is measured or estimated, Tianshu leverages a discrete event simulator to simulate the DNN model's execution. Tianshu regards each operation as an independent event that lasts for a certain time. Then, it models every computation device (like GPU) as an independent event executive, executing all events in a First-In-First-Out manner. For the communication, Tianshu abstracts the entire network as a single independent device and estimates the communication operation's execution time based on the transmission data size and the pre-defined network bandwidth.

Figure 7 shows how the simulator works.The input of the simulator is the dataflow graph of the model. Tianshu first analyzes the dataflow graph to acquire all operations' target hardware and the dependencies between them. At each iteration, it first finds all operations that their prerequisite operations have been finished. Then, it dispatches these operations to their target hardware. When an operation is finished on the hardware, the hardware will notify the simulator to update the dependent status and check whether some new operations can be executed. The simulator repeats these two steps until all operations are finished.

A notable thing is that existing works usually assume that the communication and computation operations on the same GPU will not interact with each other. However, we found that the widely used `AllReduce` operation, which synchronizes all parameters' update value among GPUs, not only transmits data but also contains some computation tasks. According to Nvidia's document, the `AllReduce` operation will occupy 16 Streaming-Multiprocessors(SM)

Model	BS	Input shape	Num of Op	Avg Op time(us)
VGG16	64	[224,224,3]	255	1013.53
Resnet50	32	[224,224,3]	1052	102.98
Inception3	32	[224,224,3]	1589	89.50
LSTM	16	[256]	62190	5.87
Seq2seq	16	[128]	30413	5.74
BERT-base	32	[32,7,30522]	6617	8.33

Table 1. Information of test DNN models. The input data is random float tensors with given batch sizes and shapes. The average operation time is measured by Tianshu on single V100 GPU.

Model	Real(ms)	Tianshu	TF-Profiler	Nvprof
VGG16	265.64	258.45	265.49	265.38
Resnet50	109.34	106.09	114.89	104.85
Inception3	142.22	136.33	151.10	134.63
LSTM	365.16	378.55	681.47	256.41
Seq2seq	174.71	188.85	310.68	88.55
BERT-base	55.10	56.52	88.76	50.17

Table 2. The sum of all operations' measuring result of six DNN models on the single V100 GPU. The result of Nvprof method is the sum all GPU kernel's execution time gathered by the Nvprof.

of the GPU in default, which will slow-down other computation tasks. For example, a Tesla V100 GPU has 80 SMs, the `AllReduce` operation will occupy 16/80 = 20% of computation resources. Tianshu simulates this mechanism by adding *N* virtual computation operation for each `AllReduce` operation (green circles in Figure 7. The execution time of these virtual operations is the same as the resource occupy proportion. In the above example, their execution time is 20% of the `AllReduce`'s execution time.

4. Evaluation Result and Discussion

In this section, we first evaluate the accuracy of Tianshu's measuring tool in Sec 4.2. Then, we evaluate the accuracy of the operation estimation model in Sec 4.3. Finally, we evaluate the discrete event simulator for full DNN models in Sec 4.4.

4.1. Experiment settings

Hardware. All experiments are conducted on two V100 servers based on Azure cloud virtual machines. Each server has 4x V100 GPU connected by PCIe gen 3 link. And these servers are connected by a 40Gbps Infiniband network.

Runtime environment. All experiments are executed in Docker containers. The Docker images uses Ubuntu 18.04 LTS, CUDA 10.0, cuDNN 7.6.5 and TensorFlow 1.15.

Test DNN models. We evaluate Tianshu using 6 well-known DNN models, including 3 CNN models: VGG16, Resnet50 and Inception3; 2 RNN models: LSTM and Seq2seq; and a transformer model: BERT-base. The input shapes and average execution time of operations are shown in Table 1. Notably, these models have different kinds of operations. CNN models contain heavy convolution operations, and the average execution time of their operations is much longer than that of other models. RNN models have many more operations, but their average operation time is very short, making it hard to measure. The BERT-base model is at the medium, with a higher average operation time than RNN models.

To prevent the interference of I/O devices, we replace the data loading part with random tensors with the same data type and shape as the original input data.

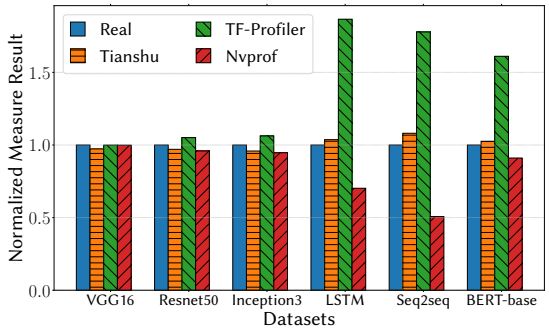

Figure 8. Measuring result of all operations on the single V100 GPU. The measuring result is normalized to the real model's iteration time to present the measuring accuracy more clearly.

4.2. Accuracy of The Measuring Tool

The measuring result of operations is the foundation of simulation. In this experiment, we compare Tianshu's measuring tool to two commonly used tools: TensorFlow's built-in profiler and the Nvprof, a GPU profiler provided by Nvidia. We use the single GPU training iteration time as the ground truth. In the single GPU scenario, all operations are executed in sequence, and the iteration time is the same as the sum of all operations' execution time. Table 2 shows the evaluation result. For Tianshu and TensorFlow profiler, the result is the sum of all operations' execution time. For the Nvprof, the result is the sum of all GPU kernels' execution time. To help readers understand the measuring accuracy, Figure 8 shows the normalized result to the real model's iteration time.

For CNN models, all tools' results are very close to the real iteration time, with a lower than 5% error. However, for RNN models that consist of minor operations, the two existing tools show huge errors. The tensorFlow profiler's measuring result could be 86.6% larger than the real value of the LSTM model because it introduces overhead to operations' execution. The Nvprof's result is also 49.3% smaller than the real value of the Seq2seq model. The reason of Nvprof's error is that it only considers the GPU kernels' execution time, not including the time to launch and scheduling GPU kernels. Notably, in the Nvprof's log of Seq2seq model, the time gap between the last and first kernel is 607ms, which is much larger than the real iteration time. This means Nvprof also introduces huge overheads between kernels' execution. Therefore, it is not applicable

Operation	Best Model	Error	No-PC Error
Add	L3N3	0.57%	11.05%
Sub	L5N6	1.47%	3.74%
Mul	L6N6	1.49%	4.48%
Realdiv	L4N3	0.66%	4.68%
Matmul	L5N7	2.92%	9.85%
Relu	L5N7	0.29%	4.52%
Conv2D	L8N12	8.94%	30.58%
Conv2DBPInput	L8N12	8.33%	32.31%
Conv2DBPFilter	L8N12	12.35%	43.07%
Average	-	4.11%	16.03%

Table 3. Average operation execution time estimation errors on V100 datasets. The value LxNy of the best model means the best prediction model contains *x* layers and *y* neurons for each layer.The column "No-PC Error" is the error of the model that is trained without the principal component. The full name of "Conv2DBPInput" operation is "Conv2DBackpropInput", same dose the "Conv2DBPFilter".

to use the time gap between adjacent kernels to represent the time use of scheduling and launching the kernel.

In conclusion, this experiment proves that Tianshu's measuring tool can produce accurate results for different types of models. Especially for models that consist of tiny operations, Tianshu's measuring tool does not introduce any measuring error and is far more precise than existing tools like TensorFlow's profiler and Nvprof.

4.3. Accuracy of operation estimation model

To evaluate Tianshu's operation estimation models, we select 9 most common ops as the target operation of the prediction. Including basic arithmetic operations, activation layer, matrix multiplication, and convolution operations. For each operation, we prepared a dataset of 11,000 samples with random input shape and parameters. Each dimension of the input tensor's shape is randomly generated within in the range of [1, 1024], and the value of other parameters are gathered from test models. 90% of data are used as the training set, and 10% of data are used as the testing set. All data is collected by Tianshu's measuring tool on a V100 GPU.

To evaluate the effect of Tianshu's principal component, we also train the estimation model with the same structure but without the principal component. Table 3 shows the prediction accuracy for all operations. On average, Tianshu's estimation model achieves 4.11% average prediction error for all these operations. However, the estimation model without the principal component has a 16.03% average error, which is three times higher than Tianshu. This result proves the effectiveness of our idea to introduce the principal component.

The value of principal component varies among different types of operations. For simple operations like basic arithmetic and activation, Tianshu achieves over 98% accuracy, while the no-PC model can also achieves a 95% accuracy. These operations' algorithm is quite simple so simple MLP models are enough to predict their execution time. For complex operations like `Conv2D` and its back-propagation operations, the principal component brings

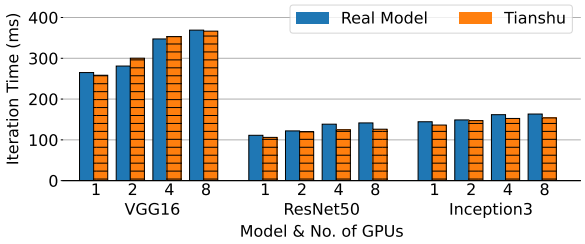

Figure 9. The simulation result of CNN models.The label of x-axis means the number of GPUs.

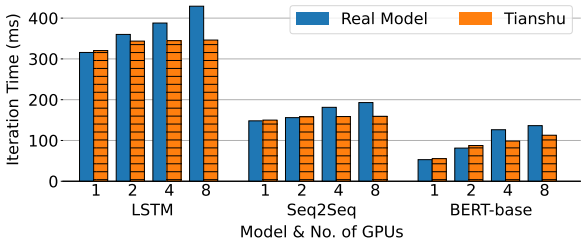

Figure 10. The simulation result of RNN and Transformer models.The label of x-axis means the number of GPUs.

huge improvement to the estimation accuracy. Without the principal component, the estimation accuracy can only achieve 60-70%. This result proves that Tianshu's principal components greatly helps the MLP model to predict complex operations' execution time.

As a reference, Habitat (Geoffrey et al., 2021) builds a dataset that contains over 600,000 samples for each operation, and uses MLP models with 8192 neurons to predict the operation's execution time on the specific target hardware using the operation's information and its measured execution time on another existing hardware. It achieves an 18% average prediction error on similar operations of the PyTorch framework. Tianshu achieves higher average accuracy with much less training data and model parameters. The reason is that Tianshu's feature engineering successfully captures helpful information about operations.

4.4. End-to-end Simulation for DNN Models

To evaluate Tianshu's discrete event simulator, we measure the end-to-end training time and the simulated time for all six DNN models on the cluster. In this experiment, we adopt the default data-parallel strategy using the Horovod library for multi-GPU training. Figure 9 and Figre 10 show the simulation result with different number of GPUs. Table 4 shows the detailed simulation error. From the result we can draw two major conclusions:

First, Tianshu makes accurate end-to-end predictions for both single GPU and distributed training. The average simulation accuracy for all models on the V100 cluster achieves 92.89%. For CNN models, the accuracy is even over 95%. This high accuracy comes from two reasons: Accurate performance measuring and estimation and the

Model	1 GPU	2 GPUs	4 GPUs	8 GPUs
VGG16	2.47%	6.84%	1.61%	0.67%
Resnet50	4.58%	1.99%	10.04%	10.88%
Inception3	5.58%	1.18%	5.71%	5.84%
LSTM	1.47%	4.55%	11.13%	19.33%
Seq2seq	1.23%	1.26%	12.51%	17.43%
BERT-base	4.67%	7.70%	21.97%	17.23%

Table 4. Simulation error of Tianshu's discrete event simulator for different DNN models with different training GPUs.

reasonable design of the simulator. If we do not consider the interaction of computation and communication operations, the simulation result will downgrade over 20% for CNN models with 8 GPUs.

Second, Tianshu achieves high accuracy for models with different architectures, including CNN, RNN, and transformers. CNN models consist of huge convolution layers and are computationally intensive. RNN models are built up with tens of thousands of minor operations, making them hard to profile accurately. Transformer model BERT-base is a communication-intensive model whose communication time (over 120ms) is far longer than its computation time.

This experiment also reveals some remaining problems. RNN models' training time grows rapidly with the number of GPUs. However, Tianshu's simulator does not capture this trend. These models contain relatively small communication data sizes, which should be overlapped by other computation operations. However, these communication operations cause considerable training time growth in the multi-GPU scenarios. This result points out that the simulation for communication still has much space to explore and improve, which will be critical to designing a more accurate simulator.

5. Conclusion

In this paper, we propose Tianshu, an accurate simulation framework for DNN models. Tianshu includes an accurate operation measuring tool, an operation estimation model to estimate the operation's execution time with unseen parameters, and a discrete event simulator to simulate the entire model's training. We evaluate Tianshu using six well-known DNN models with 8x V100 GPUs on the Azure cloud. Tianshu's measuring tools, estimation models and the discrete event simulator all achieve very high accuracy across different types of DNN models and operations. This result proves the effectiveness of Tianshu's design. The result also reveals that the communication of distributed DNN training is not as simple as we think. In the future, the simulation of DNN communication will still have a broad space to explore.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Xiaoqiang, Z. (2016). Tensorflow: A system for large-scale machine learning. In *Proc. USENIX OSDI*.

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. (2023). Gpt-4 technical report.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4171–4186.

Foley, D. and Danskin, J. (2017). Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, pages 7–17.

Geoffrey, X. Y., Gao, Y., Golikov, P., and Pekhimenko, G. (2021). Habitat: A {Runtime-Based} computational performance predictor for deep neural network training. In *USENIX Annual Technical Conference (ATC)*, pages 503–521.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *IEEE conference on computer vision and pattern recognition (CVPR)*, pages 770–778.

Jia, Z., Zaharia, M., and Aiken, A. (2019). Beyond data and model parallelism for deep neural networks. *Machine Learning and Systems*.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. (2017). In-datacenter performance analysis of a tensor processing unit. In *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. (2019). Pipedream: generalized pipeline parallelism for dnn training. In *ACM symposium on operating systems principles*.

Osawa, K., Li, S., and Hoefler, T. (2023). Pipefisher: Efficient training of large language models using pipelining

and fisher information matrices.

Pal, S. K. and Mitra, S. (1992). Multilayer perceptron, fuzzy sets, classifiaction. *IEEE Transactions on Neural Networks*, pages 683–697.

Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., and Sutskever, I. (2021). Zero-Shot Text-to-Image Generation. arXiv:2102.12092.

Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788.

Rynkiewicz, J. (2012). General bound of overfitting for mlp regression models. *Neurocomputing*, pages 106–110.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. (2019). Megatron-lm: Training multi-billion parameter language models using model parallelism.

Zaccone, G. (2016). *Getting started with TensorFlow*. Packt Publishing Birmingham.

Zhu, H., Phanishayee, A., and Pekhimenko, G. (2020). Daydream: Accurately estimating the efficacy of optimizations for dnn training. In *USENIX Annual Technical Conference (ATC)*, pages 337–352.