Fast Training of Deep Learning Models over Multiple GPUs

Xiaodong Yi
The University of Hong Kong
xdyi@cs.hku.hk

Ziyue Luo
The University of Hong Kong
zyluo@cs.hku.hk

Chen Meng
Alibaba Group
mengchen.cas@foxmail.com

Mengdi Wang
Alibaba Group
didou.wmd@alibaba-inc.com

Guoping Long
Alibaba Group
guopinglong.lgp@alibaba-inc.com

Chuan Wu
The University of Hong Kong
cwu@cs.hku.hk

Jun Yang
Alibaba Group
muzhuo.yj@alibaba-inc.com

Wei Lin
Alibaba Group
weilin.lw@alibaba-inc.com

Abstract

This paper proposes *FastT*, a transparent module to work with the TensorFlow framework for automatically identifying a satisfying deployment and execution order of operations in DNN models over multiple GPUs, for expedited model training. We propose white-box algorithms to compute the strategies with small computing resource consumption in a short time. Recently, similar studies have been done to optimize device placement using reinforcement learning. Compared to those works which learn to optimize device placement of operations in several hours using large amounts of computing resources, our approach can find excellent device placement and execution order within minutes using the same computing node as for training. We design a list of scheduling algorithms to compute the device placement and execution order for each operation and also design an algorithm to split operations in the critical path to support fine-grained (mixed) data and model parallelism to further improve the training speed in each iteration. We compare *FastT* with representative strategies and obtain insights on the best strategies for training different types of DNN models based on extensive testbed experiments.

CCS Concepts: **• Computing methodologies → Machine learning**; **• Computer systems organization → Distributed architectures**.

Keywords: Distributed training, data parallel, model parallel

Middleware '20, December 7–11, 2020, Delft, Netherlands
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8153-6/20/12…$15.00
https://doi.org/10.1145/3423211.3425675

ACM Reference Format:
Xiaodong Yi, Ziyue Luo, Chen Meng, Mengdi Wang, Guoping Long, Chuan Wu, Jun Yang, and Wei Lin. 2020. Fast Training of Deep Learning Models over Multiple GPUs. In *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3423211.3425675

1 Introduction

Deep Learning (DL) has become increasingly popular over the past years in various application domains such as computer vision, speech recognition and robotics. With increasingly complicated models and larger datasets, training of a deep neural network (DNN) model becomes an extremely time consuming job. Parallelizing the training using workers equipped with multiple GPUs or in a distributed environment is popular with current machine learning (ML) frameworks [2, 6, 9, 13].

Three of the most common parallelization strategies are data parallelism, model parallelism and pipeline parallelism. Data parallelism places a replica of the entire neural network (NN) on each device (e.g. , a GPU card) so that each device processes a subset of the training data and synchronizes model parameters in different replicas at the end of each training iteration. Model parallelism handles models with a large number of parameters, which cannot fit into the device memory, by assigning disjoint partitions of an NN each to a different device. Pipeline parallelism divides a DNN model into stages and places stages on multiple devices; it further divides each mini-batch into microbatches, so different devices can process different microbatches simultaneously.

To date, it is still not clear that given multiple GPUs, what the best strategy is to deploy a specific model onto the devices. Commonly, a practitioner may use data parallelism and replicate the model onto each GPU, but is this really always the best strategy even when a single GPU can hold the entire model? And how about large models that cannot be entirely replicated to a single GPU?

In this paper, we show that a mixture of fine-grained data and model parallelism combined with some heuristics is able to find a satisfying device placement in a fast manner with little resource consumption. We also seek to develop a software module to enable automatic model deployment without requiring model developers' code modification, which can seamlessly work with existing frameworks such as TensorFlow. For a small model which can be deployed in a single GPU, it should find a strategy achieving faster training than normal data parallelism, if there is one; for a large model which cannot be deployed entirely in a single GPU, it provides a good deployment across multiple GPUs.

We propose *FastT*, a transparent module that automatically finds and activates a satisfying deployment and execution order of operations for different kinds of models in a multi-GPU environment. Our contributions are summarized as follows:

▹ **We propose new heuristics that find deployment and execution orders in a few minutes, which are better than or as good as previous strategies that require hours to be computed.** The main reasons lie in that we extend the solution space by considering operation split and execution ordering, and we use efficient white-box heuristics rather than search or learning-based methods to reduce strategy calculating time. *FastT* is efficient enough to be executed on one node (the same as a single worker used for model training), removing the need for an additional cluster for strategy search.

▹ **We build adaptive cost models to facilitate our algorithms.** To minimize profiling overhead while obtaining accurate operation execution time on each device and inter-device communication time, we use data parallelism as the starting strategy (as long as it is feasible), try out different placements and apply linear regression to obtain the communication cost model.

▹ **We consider a larger solution space than previous approaches by considering both execution order and fine-grained parallelism within operations.** We observed significant performance variation under the same device assignment with different operation execution orders. *FastT* decides execution order and achieves fine-grained parallelism by splitting some operations on the critical path to further improve the processing speed. Experiments show that *FastT* achieves up to 59.4% speedup compared with pure data parallelism with the larger solution space.

▹ **We provide an open-source implementation of our method that transparently works with TensorFlow: developers do not have to modify their ML model to leverage our solution.** *FastT* is useful for various models, and able to automatically calculate and activate placement and execution without involving the ML developer. We have built *FastT* based on TensorFlow: once the *FastT* module is turned on, developers can transparently use it with their existing models implemented with all kinds of TensorFlow's Python APIs without modifying a single line of their code.

2 Background and Motivation

2.1 DNN Training and Parallelism

Training a DNN is an iterative process which uses a large amount of data to tune model parameters for minimizing a loss function. In current training frameworks [2, 6, 9, 13], different kinds of computation are implemented by different operations (such as Conv2D, MatMul), and input and output of these operations are called *tensors*. The computing process can typically be represented by a DAG (Directed Acyclic Graph), whose nodes are operations and edges are tensors.

Data Parallelism. The input data are partitioned to different devices. Each device shares the same model parameters. Gradients from all devices are applied to update the global model. Data parallelism can be applied in a single worker/machine with multiple GPUs [19, 26, 27], and among multiple machines [37].

Model Parallelism. The input data are sent to all devices without partition; each device is responsible for tuning a different part of the model parameters. Model parallelism is typically used for models with a large parameter size [26, 34, 45].

Pipeline Parallelism. Pipelining has been proposed to accelerate DNN training with multiple accelerators [12, 14, 22]. Many DNN models stack layers sequentially; naive model parallelism may result in only one active accelerator anytime during training. With pipeline parallelism, similar to model parallelism, different layers are deployed on different accelerators; a mini-batch is further divided into several micro-batches and these micro-batches can be processed at different layers at the same time to fully utilize all accelerators.

2.2 Fine-grained device placement

Operation-level device placement. With model parallelism, a model is typically partitioned in the layer level. A layer consists of multiple operations. To expand the solution space, some operation-level approaches are proposed [19] to decide device placement of each operation separately.

Parallelism within operations. To further extend the solution space, some studies [11, 16] investigate potential parallelism within individual operations. For example, for a Conv2D operation, it can be further parallelized by being partitioned on the batch size dimension or the channel dimension [27]. Such an approach can be regraded as fine-grain mixture of data parallelism and model parallelism according to different parallelizable dimensions of different operations.

2.3 Limitations and Challenges

Previous research [28, 46] has proposed strategies to manually optimize parallelism based on human experts' domain knowledge and intuitions. For example, Krizhevsky [28] uses data parallelism for convolutional and pooling layers and switches to model parallelism for fully-connected layers to accelerate the training of convolutional NNs (CNNs). In addition, some automated frameworks [19, 23, 27, 32] are proposed for finding efficient parallelism strategies in a limited search space. REINFORCE [32] uses a reinforcement learning method to learn efficient operation assignments on multiple GPUs. TicTac [23] explores the impact of the order of send/recv operations in distributed training. Allowing fine-grained parallelism within a single operation, FlexFlow [27] builds a new training architecture to explore the SOAP (Sample-Operation-Attribute-Parameter) search space considering parallelism within and across operations. Some other frameworks focus on specific types of networks such as CNN and RNN (Recurrent Neural Network), and provide APIs for developers to split operations by themselves such as TensorFlow mesh [7] and tofu [44].

The existing proposals have the following limitations:

First, the purpose to find optimal device placement is to save time and computing resource for a training job, and the finding process itself should not be time and resource consuming. Some existing approaches require a large amount of resources and spend a long time to obtain the strategy. For example, REINFORCE [32] and GDP [48] use another big cluster consisting of tens of workers and spend hours on learning the device placement policy.

Second, existing approaches may not be generic enough for different kinds of models or not compatible with popular training frameworks. For example, OptCNN [26] is designed for CNNs. FlexFlow implements the training framework itself, and does not support representative frameworks such as Tensorflow or MXNet.

Third, the solution space can be particularly large. Most existing studies only consider device assignment of operations, but not the execution order of operations. For example, FlexFlow defines a fine-grained search space beyond data and model parallelism, and schedules operations in the ready queue with a simple FIFO strategy.

We seek to address the following challenges in this paper:

▹ We consider not only device placement of operations, but also partitions of operations and their execution orders; the solution space becomes much larger than what the existing studies tackle. Finding a satisfying solution in a timely manner with small computation resource consumption is critical for solution adoption in a production environment.

▹ It is easier to design strategies for a specific type of models. However, a generic approach needs to analyze the structure (DAGs) of different models and builds the respective cost models.

▹ Since ML developers may use various APIs to implement their models (even when they are using the same ML framework such as TensorFlow), it is hard to design a unified software module to transparently support their existing models without modification.

A practical model deployment and execution module must be fast, light-weight, generic and compatible with existing training architectures at the same time.

3 Problem Definition

We first formally define the problem we intend to solve. The objective is to find a good device placement and execution order to achieve parallelism across operations in a DNN model, and also identify potential operations which can be partitioned into several sub-operations to achieve fine-grained parallelism within individual operations.

The input of the problem includes: (a) the DAG computing graph, (b) the set of devices (GPUs) and memory limitation of each device, and (c) the cost models for computation and communication. The computation cost model provides computing time of a given operation on a specific device, and the communication cost model gives inter-device tensor communication time of adjacent operations running on different devices.

The output solution consists of three parts: (i) a partition list of operations which should be partitioned (each item in the list has three elements, the operation's name, partition dimension, and the number of partitions); (ii) device placement of each non-partitioned operation and each sub-operation (due to splitting operations in the partition list); and (iii) execution order of operations and sub-operations.

It should be noted that we focus on NN whose computing graph is a DAG. Some networks can be implemented as a graph with cycles in TensorFlow, e.g. a dynamic RNN which includes a while loop, and whether to exit from the loop is decided during runtime. For such a model, we optimize exection of the DAG within each of its loops.

The problem of deciding execution order and placement of a DAG with unit operation execution time is known as the single execution time scheduling problem, which is NP-complete [42]. Our problem poses even greater challenge as we assume heterogeneous operation execution time. Therefore, we propose efficient heuristic algorithms in Sec. 5 to find a good solution of our problem.

4 System Design

FastT is built based on TensorFlow, addressing parallelism both within and across operations in a DNN model. It calculates both device placement and execution order for each operation in the computation graph, with operations potentially further partitioned.

Fig. 1 illustrates how *FastT* fits into the architecture of TensorFlow, and colored blocks represent components we

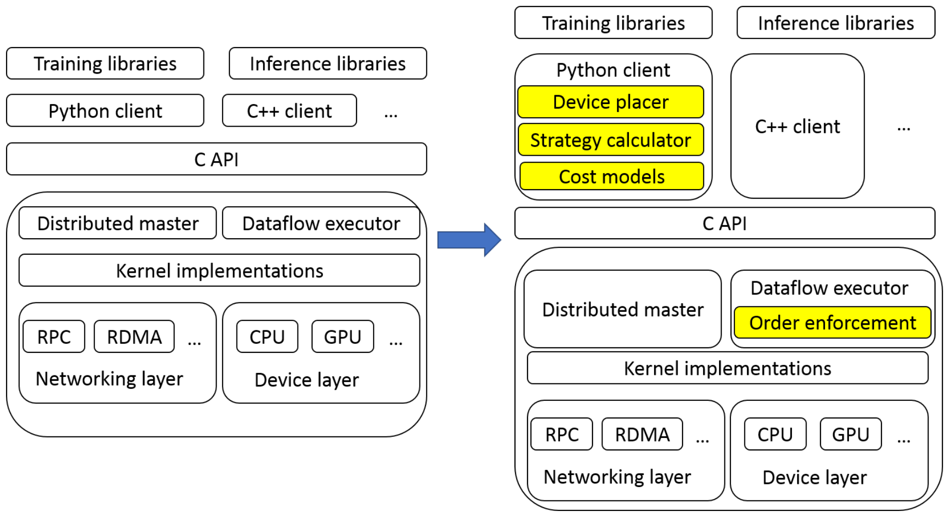

Figure 1. Architecture of TensorFlow (left) and *FastT*(right)

implement for *FastT*. The *strategy calculator* computes device placement and execution order for the current model using the algorithm to be introduced in 5.2. The *device placer* assigns different devices to run different (sub-)operations according to the strategy computed by the strategy calculator. In the dataflow executor, *order enforcement* is responsible for organizing the execution order of (sub-)operations. The *cost models* component records the execution time of (sub-)operations executed on different devices and the data transmission time when adjacent operations are handled on different devices.

The workflow of *FastT* is as follows: *Initially*, *FastT* requires several pre-training steps to bootstrap the cost models, which uses different placement strategies to run the DNN model to update its cost models. To activate a new strategy computed with the updated cost models, the training session does checkpoints of current model parameters, and restarts to create a new graph based on the operation partition lists; then the device placer activates the device placement and the order enforcement module enforces the execution order of (sub-)operations. The training session then restarts with restored parameters from the checkpoints. After a new strategy is activated, *FastT* records the per-iteration model training time; if it finds that the per-iteration execution time with the new strategy is even longer than the previous one, it rolls back the strategy to the previous one. When the cost models become stable (the average time of the same (sub-)operation(s) on the same device(s) does not vary much), we finish the pre-training stage. *Afterwards*, the model is trained normally using placement and execution order strategies computed by the strategy calculator, and the cost models are updated only when the execution times have changed significantly based on our periodical profiling.

Currently, we use checkpointing and restart the model for activating a new strategy, since commonly adopted TensorFlow versions do not allow modification of a graph structure when the session has already run the graph. After the normal training stage starts, the cost models are not updated often.

Cost Models. The computation cost model provides the execution time of a (sub-)operation on a device, using the operation's name and device as the key. The communication cost model provides the tensor transfer time between adjacent operations assigned to two different devices, according to tensor size and device pair. Different from simulation-based measurement of such time [27], we profile the training process and record real execution/transmission time, based on the RunMetadata [5] generated by the TensorFlow profiler.

During the pre-training stage, *FastT* first uses its algorithm (DPOS in Sec. 5) to compute device placement and execution order strategies (a default data or model parallel strategy is used when cost models are empty), trains the DNN model with these strategies for several iterations (aka steps), and profiles the training process to update the cost models. Especially, when our algorithm finds a cost that it needs (e.g. , execution time of an operation on a device) is not in the cost model, it sets the cost to 0, so that the algorithm prefers to explore the placement, and then the profiler can obtain the real cost of this placement in the following training steps. It typically only takes several iterations to obtain the complete computation cost model, considering that we use data parallelism as the starting strategy (as long as the model can be fit into a GPU) by which each operation is replicated to different GPUs and their execution time on different devices is learned. For a large model that cannot be fit into a GPU, we use model parallelism, try different placements on multiple GPUs, and obtain the cost models.

To build the communication cost model, we gather tensors across the same source-destination device pairs into one group. For each group, we use linear regression to obtain a linear model: tensor size vs. transfer time. In each update of the cost model, newly collected data are fed and parameters of the linear model are re-computed. The models capture available bandwidth and potential congestion along each device-device path.

Strategy Calculator. It is the key component to carry out the algorithms that we will discuss in Sec. 5. During the pre-training stage, it calculates device placement, execution order and operation partition lists, and obtains the cost models. During the normal training stage, it periodically activates the profiler, updates the cost models, and recalculates new strategies. If the estimated per-iteration training time with the new strategies (among output of our DPOS algorithm) is smaller than that of previous strategies, the new strategies are activated.

Device placer. Device placer is responsible for assigning each operation onto a device (GPU) according to the strategy computed by the strategy calculator.

Order Enforcement. After obtaining execution order of (sub-)operations from the strategy calculator, the enforcement module sets the indices of (sub-)operations in the order list as their priorities, and enforces the execution order in TensorFlow's executors.

5 Operation Placement and Ordering Heuristics

5.1 Listing Scheduling

We design a listing scheduling method to compute the device placement and execution order of operations, inspired by algorithms handling DAG task scheduling over multiprocessors [20, 41]. With listing scheduling, the whole solution space is reduced in two phases: (i) operation prioritization for deciding device placement sequence of all operations, and (ii) device selection which chooses the operation in the order of their priorities and assigns the best device to each selected operation, to minimize the operation's finish time.

Operation Prioritization. The priority decides the device placement sequence, which is slightly different from the execution order of operations. We exploit a critical-path [41] based heuristic for computing a rank $rank_u$ for each operation o_i in the DAG:

$$rank_u(o_i) = w_i + \max_{o_j \in succ(o_i)} (c_{i,j} + rank_u(o_j))$$

where w_i is the maximal execution time of operation o_i (over different devices that it could be assigned to run), $succ(o_i)$ is the set of immediate successor operations of o_i, and $c_{i,j}$ is the maximal transmission time of the tensor from operation o_i to operation o_j (over different device pairs that they can be located on). $rank_u(o_i)$ represents the length of the critical path from operation o_i to the exit operation, and can be computed recursively by traversing the computation graph, starting from the exit operation. The rank of the exit operation is: $rank_u(o_{exit}) = w_{exit}$.

We use $rank_u(o_i)$ as o_i's priority, such that the next operation to be placed is always the entry operation in the new critical path of the current sub-graph, excluding the operations that has already been considered.

Device Selection & Execution Order. We use $EST(o_i, d_j)$ and $EFT(o_i, d_j)$ to represent the earliest execution start time and the earliest execution finish time of operation o_i on device d_j, respectively. For the entry operation o_{entry} of the DAG, we have

$$EST(o_{entry}, d_j) = 0 \qquad for\ d_j \in set\ of\ devices$$

For other operations, EFT and EST can be computed starting from the entry operation as follows:

$$EST(o_i, d_j) = \max\{avail[j], \max_{o_m \in pred(o_i)} (EFT(o_m) + \hat{c}_{m,i}^{d_j})\}$$

$$EFT(o_i, d_j) = w_{i,j} + EST(o_i, d_j)$$

Here, $avail[j]$ is the earliest available time of device d_j; $pred(o_i)$ is the set of immediate predecessor operations of o_i; $\hat{c}_{m,i}^{d_j}$ is the actual tensor transmission time between o_i's immediate predecessor o_m and o_i, if o_i is assigned to device d_j. Note that $avail[j]$ is not the time when d_j completes the execution of its last assigned operation: it is possible for our algorithm to insert an operation into an earliest idle time slot between two already-scheduled operations on a device; the length of the idle time slot should be sufficient to execute this operation, and inserting the operation into this idle time slot should preserve precedence constraints; $avail[j]$ is the start time of such a timeslot. We use $ST(o_i)$ and $FT(o_i)$ to represent the actual execution start time and execution finish time of operation o_i.

Our algorithm aims to minimize the overall actual execution time of operations on the computation graph's critical path (based on their placement), which is the lower bound of the end-to-end execution time of the DAG (there could be gap time between operation executions). To compute the critical path, the entry operation is selected, and then we recursively select the operation with the largest rank among the successors of the previous operation.

We consider operations in the DAG according to the order computed. If the operation is on the critical path, assign it to a *critical-path device*. We choose a critical-path device as follows: for each available device, we simulate placing as many remaining operations on the critical path as possible onto the device (within its memory capacity), and compute the average execution time of the operations on the device using values from the computation cost model; we choose the device with the smallest average time as a critical-path device. If an operation is not on the critical path, we assign it

Algorithm 1 Device Placement and Operation Sequencing (DPOS)

```
1:  Input: Graph G(O,E); Device Set D; Computation Cost Model $C_{comp}$; Communication Cost Model $C_{cmmu}$;
2:  Output: New Device Placement Strategy $S_{new}$; Execution Order List A[]; Finish Time of Exit Operation $FT(o_{exit})$.
3:  Set $w_i$ to be the max execution time of operation i and $c_{i,j}$ to be the max communication time between operations i and j .
4:  Compute $rank_u$, critical path $SET_{CP}$.
5:  Select a device set $d_{CP}$ to place operations in critical path based on average computation time and memory capacity.
6:  Create priority queue L for operations by decreasing order of $rank_u$ values.
7:  while L is not empty do
8:      $o_i \leftarrow L.dequeue()$
9:      if $o_i \in SET_{CP}$ then
10:         $S_{new}[o_i] = d_{CP}(o_i)$
11:     else
12:         for d in D do
13:             if memory need of $o_i$ exceeds capacity of $d$ then
14:                 $EFT(o_i, d) \leftarrow +\infty$
15:             else
16:                 Compute $EFT(o_i, d)$
17:             end if
18:         end for
19:         $S_{new}[o_i] = \arg\min_{d \in D} EFT(o_i, d)$
20:         $FT(o_i) = EST(o_i, S_{new}[o_i])$
21:     end if
22: end while
23: Compute Execution list $A$ by sorting operations in ascending order of $ST(o_i)$
24: Compute $FT(o_{exit})$=$EFT(o_{exit}, S_{new}[o_{exit}])$
25: Return: $S_{new}$, $A$, $FT(o_{exit})$
```

to another device with sufficient memory which minimizes the EFT of the operation. During operation-device assignment, when a critical-path device's memory is full, we find another critical-path device and assign as many critical-path operations to it as possible.

Our Device Placement and Operation Sequencing (DPOS) algorithm is given in Alg. 1. We identify the following properties of DPOS. We use ω_{DPOS} to represent the end-to-end processing time of the DAG. The time intervals in $[0, \omega_{DPOS}]$ can be categorized into two exclusive sets A and B: A includes all time intervals when all the devices are busy, and B includes intervals when at least one device is idle. If $B = \varnothing$, DPOS is obviously optimal. Thus, we focus on the case where $B \neq \varnothing$. We assume B is the union of N intervals: $B = [b_1^l, b_1^r] \cup [b_2^l, b_2^r] \cup \ldots \cup [b_N^l, b_N^r]$ with $b_1^l < b_1^r < b_2^l < \cdots < b_N^r$. We use O to represent the set of all operations in the DAG.

Lemma 1 There exists a chain $X : o_{i_M} \rightarrow o_{i_{M-1}} \rightarrow \ldots \rightarrow o_{i_1}$ in O that covers B, if the memory capacity of devices is sufficient to host operations assigned. That is, the total execution time plus maximal overall data transmission time along chain X is no less than the total duration of B: $\sum_{n=1}^{N}(b_n^r - b_n^l) \leq \sum_{m=1}^{M} w_{i_m} + \sum_{m=1}^{M-1} c_{i_m,i_{m+1}}$.

Theorem 1 The end-to-end processing time of the DAG, ω_{DPOS}, satisfies: $\omega_{DPOS} \leq 2\omega_{opt} + C_{max}$, where ω_{opt} is the optimal DAG execution time in an ideal system without tensor transmission time, and C_{max} is the maximal overall data transmission time along any chain in O.

The detailed proofs are given in the Appendix.

5.2 Operation Splitting

The DAG execution time may be further reduced by splitting operations on the critical path into sub-operations, for further parallelism to reduce the overall execution time of the critical path. Different types of operations have different dimensions to be split. For example, Conv2D can be partitioned on the batch size dimension for fine-grained data parallelism within the operation, and also on the channel dimension to achieve fine-grained model parallelism. Splitting operations does not change training semantics through graph modification, hence resulting in no model accuracy loss. We propose our second heuristic OS-DPOS (Operation Splitting Device Placement and Operation Sequence) to perform operation splitting based on DPOS.

The input graph to Alg. 2 is decided as follows: if the model is too large to be fit into a single device, we input the DAG of the model; otherwise, we construct a data parallel graph based on the model DAG as the input, where the model is replicated as many times as the number of devices (i.e., we adopt data parallelism as our start deployment strategy in order for the algorithm to identify a better strategy beyond pure data parallelism). In the algorithm, a function SplitOperation is invoked to generate the updated graph when an operation is split on a specific dimension with a certain split number. As an example, here we only show one split method which is suitable for some types of operations (e.g. , suitable for Conv2D and not for BatchNorm). Different split methods are available for splitting other types of operations [7, 26, 27].

The algorithm first invokes Alg. 1 to compute an initial device placement and execution order. Then it calculates the new critical path based on the placement strategy and splits the operations along the critical path in descending order of their computing time. For a specific operation, Alg. 1 is called to compute the corresponding critical path, device placement and execution order after splitting it on each

Algorithm 2 OS-DPOS

```
1: Input: Graph $G(O,E)$; Device Set D; Computation Cost Model $C_{comp}$; Communication Cost Model $C_{cmmu}$;
2: Output: Operation Split List $SP[]$; New Device Placement Strategy $S$; Execution Order List $A$;
3: Compute $S_{new}$; $A[]$; $FT_{old}(o_{exit})$ using DPOS(G,D, $C_{comp}$,$C_{cmmu}$).
4: Compute Critical path (CP) based on $S_{new}$ and G.
5: sort CP by descending order of computation time.
6: Initialize $SP \leftarrow []$;$G_{init} \leftarrow G(O,E)$;$S \leftarrow S_{new}$
7: for operation $op$ in CP do
8:     With different $d \in$ $parallelizable\ dimensions$ and n$\in$ # of GPUs(D), call DPOS(SplitOperation($G_{init}$, $op, d, n$), D,$C_{comp}$,$C_{cmmu}$,$S$) and record the smallest $FT(o_{exit})$ and corresponding dimension $d$, split num $n$, $S_{new}$ and $A_{new}$.
9:     if $FT(o_{exit}) < FT_{old}(o_{exit})$ then
10:        Update: $FT_{old}(o_{exit}) \leftarrow FT_{new}(n_{exit})$, $G_{init} = G_{new}$, $S \leftarrow S_{new}$, $SP \leftarrow SP \cup (op,d,n)$, $A \leftarrow A_{new}$
11:    else
12:        break
13:    end if
14: end for
15: Return: $SP$, $S$, $A$.
16: function SplitOperation (Graph:G(O,E), Operation:op, Dimension:d, Split num:n)
17:    for $i \leftarrow 1,2,...,n$ do
18:        Create new sub-operation $s_i$
19:    end for
20:    for operation $pre \in predecessors(op)$ do
21:        add a split node $sp$ and connect it to the $n$ partitions split from edge $(pre, op)$ on dimension $d$: $p_1, p_2, ..., p_n$.
22:        connect $p_i$ to $s_i$.
23:    end for
24:    for operation $suc \in successors(op)$ do
25:        add a concatenate node $con$ that concatenates $s_1, s_2, ..., s_n$.
26:        connect $con$ to $suc$.
27:    end for
28:    Remove operation $op$ and edges connecting to it.
29:    Return: Updated graph $G_{new}$
30: end function
```

dimension and with each split number, and the best split of the operation which achieves the smallest FT of the exit operation in the DAG is identified. Only if this time with the best split is smaller than before splitting, the algorithm records the corresponding best split dimension and split number, and adds the decisions to the split list; otherwise, the algorithm stops the loop and no longer explores the remaining operations on the critical path.

It is noteworthy that *FastT* may not use all the input devices, and can choose a subset which achieves better performance than using all. Strategy calculator in *FastT* carries out Alg. 1 to derive device placement and execution order of all (sub-)operations.

6 Implementation and Evaluation

6.1 System Implementation

We implement *FastT* over TensorFlow 1.14.

Strategy Calculator is built in Python client of TensorFlow (1660 LoC in Python). We add the control logic inside the initialize function and run function of class BaseSession. It is the entry point to invoke TensorFlow C++ core runtime from Python, and most high-level Python APIs are based on the BaseSession class. Therefore, model developers can transparently use our module with their existing models. The strategy calculator activates TensorFlow profiler for updating the cost models and computes new strategies in the run function of BaseSession using a single CPU core.

Device Placer is simple module implemented with 20 LOC in Python. It first checks the co-location constraints of operations and then uses built-in functions of TensorFlow to implement the device placement. When the training is done over multiple machines, we use in-graph [4] implementation so that a single global computation graph can be placed on these machines.

Cost Model. We extend TensorFlow internal tracer to fetch the raw meta-data of each operation during the training process (198 LOC in Python), for building the cost models.

Order Enforcement is implemented within the executors in TensorFlow C++ runtime (107 LOC). By default, the runtime scheduler executes the operations in the ready queue following FIFO (First-In-First-Out). We set each operation with a priority according to the execution order computed by the strategy calculator, and schedule operations according to their priorities. We used to directly add control dependency to enforce execution order, which adds strong constraints in the graph, loses the chance for further optimization (such as the graph pruning by TensorFlow), and sometimes leads to poor performance. We hence exploit the priority-based method to provide the scheduler more flexibility, while satisfying control dependencies.

Since we directly modify the code in Tensorflow's `Session.run` function to take over the control of all following processing, the developers do not need to change a single line of their model code, when using the TensorFlow framework compiled with our modules.

6.2 Evaluation Methodology

Testbed setup. We deploy *FastT*-boosted TensorFlow framework in physical machines, each equipped with 8 NVIDIA Tesla V100 GPUs with NVLinks, where each GPU has 16GB

Table 1. Training speed (samples/s) of models using different strategies with strong scaling. DP represents Data Parallel. The last column shows the speed-up of FastT with the best baseline performance and the results corresponding to this speed-up are bold.

Models(global batch size in samples)	1 GPU	2GPUs		4GPUs		8GPUs		8GPUs (2servers)		Speedup
		DP	*FastT*	DP	*FastT*	DP	*FastT*	DP	*FastT*	
Inception_v3(64)	191.0	326.5	323.2	**467.1**	**474.1**	432.4	438.3	378.7	415.6	1.5%
VGG-19(64)	129.0	149.5	199.4	**184.9**	**294.9**	126.9	132.5	110.7	122.3	59.4%
ResNet200(32)	89.3	114.2	**142.2**	**122.1**	132.2	88.4	91.1	77.4	82.6	16.4%
LeNet(256)	8827.5	14222.2	**23272.7**	17006.6	19692.3	**17066.6**	19692.3	13473.6	16000.0	36.3%
AlexNet(256)	1630.5	1868.6	**2752.6**	**2000.0**	2534.6	1695.3	1729.7	1391.3	1542.1	37.6%
GNMT(4 layers)(128)	301.1	435.3	479.4	573.9	**636.8**	**584.4**	606.6	458.7	455.5	8.9%
RNNLM(64)	345.9	**349.7**	**395.0**	335.0	345.9	254.9	273.5	132.5	131.1	12.9%
Transformer(4096)	7613.3	11221.9	11346.2	**13518.1**	**15515.1**	5244.5	5258.0	4586.7	4807.5	14.7%
Bert-large(16)	84.2	115.9	132.2	**124.0**	**152.3**	101.2	117.6	82.9	98.7	22.8%

Table 2. Training speed (samples/s) of models using different strategies with weak scaling. The last column shows the speed-up of FastT with the best baseline performance and the results corresponding to this speed-up are bold.

Models(batch size per GPU in samples)	1 GPU	2GPUs		4GPUs		8GPUs		16GPUs (2servers)		Speedup
		DP	*FastT*	DP	*FastT*	DP	*FastT*	DP	*FastT*	
Inception_v3(64)	195.1	375.3	375.3	695.6	695.6	1245.7	1340.3	**2211.6**	**2316.7**	4.7%
VGG-19(64)	130.3	240.6	255.4	475.8	504.9	707.1	819.2	**1155.7**	**1378.2**	19.2%
ResNet200(32)	90.6	175.8	178.7	322.4	346.89	598.1	608.0	**942.9**	**1001.9**	6.2%
LeNet(256)	9142.8	16516.1	18285.7	20897.9	**24975.6**	**21557.8**	23011.2	18533.9	22021.5	15.8%
AlexNet(256)	1600.0	2508.9	2994.1	2708.9	**3112.4**	2756.3	2904.9	**2848.4**	2890.6	9.3%
GNMT(4 layers)(128)	308.4	571.4	606.6	1047.0	1101.0	1988.3	1980.6	**3136.2**	**3292.6**	4.9%
RNNLM(64)	353.5	592.5	695.6	898.2	930.9	964.2	1013.8	**1109.4**	**1140.3**	2.7%
Transformer(4096)	7861.8	15142.3	15170.3	26815.0	28151.2	47976.5	50334.9	**73388.6**	**73388.6**	0%
Bert-large(16)	81.6	137.3	146.1	229.3	248.0	361.5	421.0	**531.1**	**572.7**	7.8%

memory, and 2 Intel(R) Xeon(R) Platinum 8163 CPUs, where each CPU has 24 cores.

Benchmark models. We experiment with 5 CNN models (VGG19 [39], ResNet200 [24], AlexNet [29], LeNet [1] and Inception-v3 [40]) and 4 NMT models (Transformer [43], Bert-large [17], GNMT [46] and RNNLM [47]).

Baseline strategies. We use data parallel (DP) strategies and results of REINFORCE [32], GDP [48], FlexFlow [27] and Post [18] as baselines. For data parallelism, we adopt default data parallel implementation in TensorFlow slim [3], and compare the performance under both strong scaling (which retains the same global batch size when the number of GPUs varies) and weak scaling (which retains a fixed batch size at each GPU). For REINFORCE, GDP, FlexFlow and Post, we compare the strong scaling performance with results extracted from their papers (they all adopt strong scaling): REINFORCE, GDP and Post need tens of servers to compute their policies; the available source code of Flexflow only includes the part of applying a given strategy but not the code for running their search method to find the strategy, and is hence not directly usable for experimental comparison.

We use training speed (samples/second) rather than the per-iteration training time as the performance metric, because in weak scaling, the global batch size grows with the number of GPUs, and as a result per-iteration times cannot be directly compared. Since our method preserves the semantics of model training, and does not change the number of iterations to converge for each model, we do not show the total iteration number in our evaluation. In strong scaling, we choose the global batch size to fully utilize a single GPU to ensure no out-of-memory (OOM) when using only one GPU for training; in weak scaling, we choose the per-GPU batch size to fully utilize a single GPU without incurring OOM. All our results are averaged over 500 iterations after a warm-up of 10 iterations.

6.3 Performance of *FastT*

Per-iteration speed-up. In Table 1, we see that with strong scaling, *FastT* outperforms default data parallelism in most cases, and achieves up to 59.4% speed-up when training VGG using 4 GPUs. With more GPUs (e.g., 8), the performance

Table 3. Per-iteration training time (in seconds) when training Bert-large with DP and *FastT*. OOM is out of memory.

Models (global batch size)	Single GPU	2GPUs	
		DP	*FastT*
Bert-large(16)	0.192	0.138	0.121
Bert-large(32)	OOM	0.233	0.219
Bert-large(40)	OOM	OOM	0.287
Bert-large(48)	OOM	OOM	0.316

of both strategies may degrade due to more communication overhead among model replicas and smaller batch size per GPU which cannot achieve good GPU utilization, but *FastT* still does better. In the case of 8 GPUs (2 servers), we experiment in a distributed setting with 4 GPU cards each on two servers, and include inter-server communication time into our cost models. The improvement of *FastT* over the default strategy is in general better in this distributed setting, than with all 8 GPUs on the same server. This is because the default strategy performs worse in a multi-server setting than on the same server, while *FastT* can find better solutions by capturing the communication overhead across servers using the communication cost model.

With weak scaling, the performance of data parallelism in Table 2 is similar to the performance reported in DAWNBench [15] and NVIDIA Benchmark [8]. We see that *FastT* still performs better than data parallelism, and a 19.2% speed-up when training VGG in the case of 16 GPUs (2 servers). As compared to the speed-up in Table 1, the improvement over data parallelism is smaller, which is because the utilization of each GPU with data parallelism is much higher than in the strong scaling setting, leaving us a much smaller optimization space by moving operations around across the devices.

We observe that the improvement with *FastT* is better with Bert-large than Transformer. The Transformer model can be fit into a single GPU with the standard batch size, so that data parallelism performs pretty well already. When training Bert-large, the batch size per GPU is much smaller, as otherwise out-of-memory (OOM) errors occur; hence training Bert-large with data parallelism may not do well due to the under-utilization of GPU computation capacity with the small batch size. On the other hand, *FastT* can find better solutions of placing most operations in the model in one GPU, to better utilize GPU computation capacity while minimizing inter-GPU communication.

Unless otherwise stated, our following experiments are based on strong scaling, and the global batch size used to train a model is the same as indicated in Table 1.

Support larger batch size for very large models. For bert-large, its model cannot be fit in a single GPU when batch size is larger than 16. We set the maximal sequence lengths in bert models to be 64. Table 3 shows that with *FastT*, we can efficiently exploit 2 GPUs to train it with larger global batch sizes (e.g. , 48), while data parallelism can only support global batch size of 32. In addition, developers do not need to worry about manual placement of such a large model between devices.

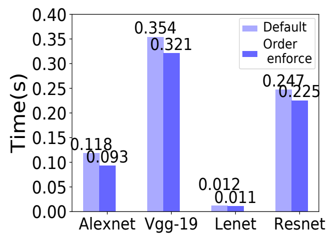

Figure 2. Performance gain of order enforcement.

Table 4. Time (in seconds) to run Alg. 2

Models(global batch size)	2GPUs	4GPUs	8GPUs
Bert-large(32)	448.9	470.3	529.9
Inception_v3(64)	28.7	64.5	124.8
Vgg-19(64)	24.41	62.74	118.4
Resnet200(32)	201.2	481.9	792.5
Lenet(256)	3.54	8.71	11.28
Alexnet(256)	4.23	9.58	18.46
Transformer(4096)	783.0	1952.6	5775.2
GNMT(128)	122.31	259.43	522.85
RNNLM(64)	48.95	92.31	174.22

Order Enforcement. We evaluate the performance gain brought by operation execution ordering, and compare with TensorFlow's default execution order. In TensorFlow, the executor chooses operations from a ready queue using FIFO. In Fig. 2, each model is run using 2 GPUs. We see that per-iteration time is reduced by up to 26.9% when order enforcement is enabled.

Time for strategy calculation. Table 4 shows the time needed to compute placement and execution order with Alg. 2 in *FastT*, which is within several minutes for most models. It takes more than 1 hour to compute the strategies for deploying the Transformer model over 8 GPUs, due to the very large number of operations in the model. Besides, the strategies are computed through real model training, such that the strategy search time includes profiling time and system restart time (for activating changed strategies). Still *FastT* can compute the strategies using much less time and resources than existing approaches such as REINFORCE and GDP.

6.4 Comparison with other strategies

We next compare the speed-up of *FastT* with REINFORCE, GDP, FlexFlow and Post. We use strong-scaling data parallelism as the baseline, and show each strategy's processing speed divided by that of the data parallel strategy in Fig. 3.

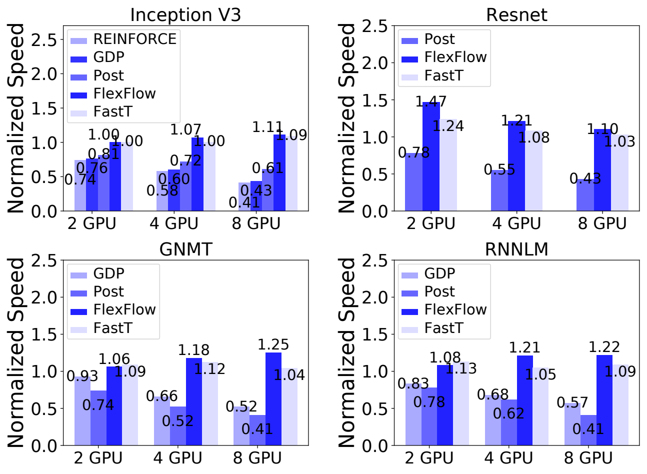

Figure 3. Performance comparison among *FastT*, REINFORCE, GDP, FlexFlow and Post

The models being evaluated are those with results in the respective papers as well. *FastT* outperforms REINFORCE, GDP and Post in all respective cases, as REINFORCE, GDP and Post do not consider data parallelism and operation split, and hence their solution spaces are limited. FlexFlow may find a better solution than *FastT*, due to its larger solution space and extensive search-based algorithm to find the strategy. However, *FastT*'s performance is close; being compatible with TensorFlow, it is more generally usable. Further, the time complexity of *FastT* is linear with the number of operations and devices, while the search space in FlexFlow increases exponentially with the increase of operations and devices.

6.5 Analysis of result placements

Operation placement. Fig. 4 shows the number of operations assigned to each GPU with *FastT*. Different from pure data parallelism that assigns model replica to each GPU, *FastT* does not always allocate operations evenly among GPUs. In the case of 4 GPUs, one GPU has many more operations while the numbers on others are pretty even. Our investigation shows that replicas of operations with large parameters are placed in one GPU rather than 4 GPUs, to avoid inter-GPU aggregation of gradients of these parameters during training. For other computation-intensive operations, they are evenly placed onto 4 GPUs to reduce end-to-end processing time, which implies that the computation time saving due to data parallelism exceeds the cost for aggregating gradients across 4 GPUs for these operations.

Operation split. Table 5 shows the split decisions for some representative operations in Vgg-19, as made by *FastT*, together with their execution time (before splitting) and parameter sizes. We can see that in Vgg-19, some conv operations have longer execution time than others, so they are most likely to be split. Fc operations with large parameter sizes

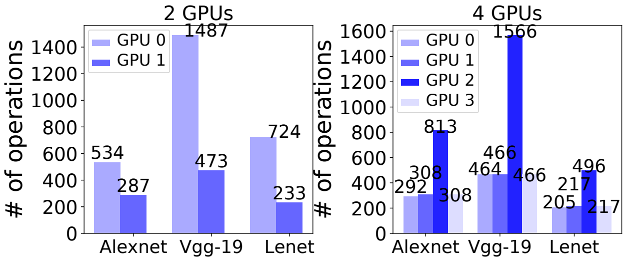

Figure 4. Number of operations in each GPU using *FastT*

Table 5. Split decision for representative operations in Vgg-19.

Operation	Time(ms)	Weight(KB)	Split
Conv1_1	1.847	1.792	False
Conv1_2	11.14	36.928	True
Conv1_2bp	26.744	36.928	True
Relu1_2	1.08	0	False
Pool1	0.737	0	False
Fc6	1.374	102764.544	False

are not split, to avoid overhead of broadcasting parameters to all replicas. Operations being split usually have longer execution time and smaller parameter size, to strike a good trade-off between computation performance gain and extra communication overhead incurred by the split.

Table 6 compares model training performance when we enable operation split in *FastT* and not. The experiments are done under the settings achieving the best speedup as in Table 1. We see that with CNN models such as Inception, Vgg and ResNet, Conv2D and Conv2Dbp are the key operations whose splits bring performance gain. However, for LeNet and AlexNet, these operations are not split due to small input tensor sizes to them (such that these operations' computation time is small). Further, operations in LSTM-based NMT models (GNMT and RNNLM) are not split because no computation intensive operation is found. For attention-based models (Transformer and Bert-large), MatMal operations are split, which are the most computation-intensive operations in these models.

Time Breakdown. We show the average computation time and memory copy time (i.e., tensor transfer time) when training the models using pure data parallelism and *FastT* on 2 GPUs in Fig. 5. Due to overlap of computation and memcpy (communication), the overall per-iteration training time is usually not equal to the sum of computation and memcpy time. We observe that even though the computation time with *FastT* is increased, its memcpy time and per-iteration time are reduced. Main reasons are as follows. With data parallelism, operation replicas require gradients from other replicas in each iteration, which involves memory copy since the replicas are assigned to different GPUs. *FastT* can reduce

Table 6. Per-iteration training time (in seconds) with/without operation split.

Models	No split	Split	Speedup	Key split op
Inception_v3	0.161	0.154	4.54 %	Conv2D,Conv2Dbp
Vgg-19	0.356	0.321	10.91 %	Conv2D,Conv2Dbp
ResNet200	0.249	0.225	10.67 %	Conv2D,Conv2Dbp
LeNet	0.011	0.011	0 %	None
AlexNet	0.093	0.093	0 %	None
GNMT	0.201	0.201	0 %	None
RNNLM	0.162	0.162	0 %	None
Transformer	0.281	0.264	6.44 %	MatMul
Bert-large	0.113	0.105	7.62 %	MatMul

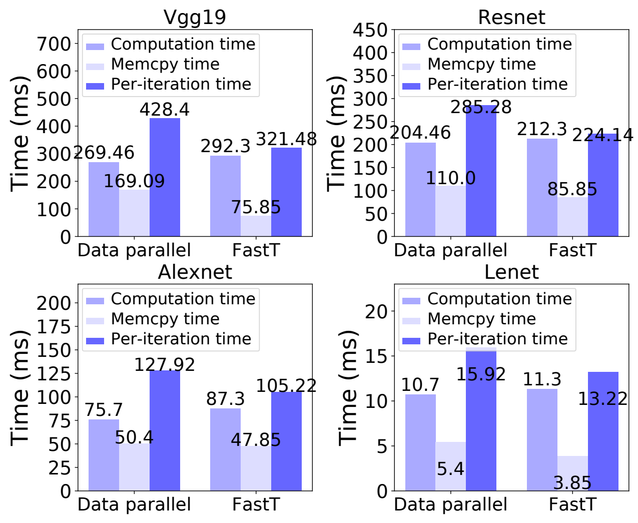

Figure 5. Average computation and memcpy time per iteration

memcpy cost by assigning some replicas of an operation to the same GPU (as validated by its uneven operation assignment among all GPUs), which on the other hand may increase GPU time due to processing more operations on some GPU.

7 Related Work

Device Placement for Deep Learning Models. Researchers have been seeking the best placement strategy to assign operations in a DNN to different devices, to minimize execution time of the computation graph. The Google team used reinforcement learning to tune a placement strategy [32]. Some follow-up work propose more advanced algorithms to reduce learning time for deriving the policy [19, 21, 30], enlarge the solution space for better strategies [12, 26, 27, 44], optimize the reward function and sampling methods [18, 19, 33], or learn a more general model applicable to different computation graphs [10, 35, 36, 48]. For example, Placeto [10], GDP [48] and REGAL [35] use GNNs to generalize their models so that they can handle unseen computation graphs, and REGAL further considers the execution order of operations; however, these proposals only consider model parallelism of computation graphs, so the performance is limited. All the above studies treat the placement problem as a black box, and usually require hours of learning to obtain a satisfying policy, using large amounts of computing resource for policy training. Stanza [45] separates CONV layers and fully-connected layers into different workers to reduce communication overhead; it only optimizes these two types of layers. DLPlacer [34] studies hybrid data and model parallelism, but its device placement is based on a subgraph of the model rather than the entire graph.

Fine-grained parallelism within operations. For neural networks such as a CNN, the fully connected layer is much larger than others; operations in that layer can be partitioned into several small sub-operations, and sub-operations can be assigned to different devices to reduce the execution time along the critical path. Alex [28] uses data parallelism for convolutional and pooling layers and switches to model parallelism for densely-connected layers to accelerate CNNs. TensorFlow mesh [38] provides high-level APIs for developers to specify parallelizable dimensions for different kinds of operations. They depend on developers to manually decide the parallelism strategy, which requires lots of experience. Tofu [44] utilizes a partition-n-reduce method to split a single operation into fine-grained operations, and a dynamic programming method to recursively optimize the partition. It does not consider device placement of operations. OptCNN [26] parallelizes CNN models by splitting operations along batch and channel dimensions; it does not consider parallelism across different operations.

Pipeline parallelism for DNN training. Chen et al. [14] use a pipelining strategy to update models with delayed data and allow to compute different layers concurrently. Wu et al. [46] accelerate computation of RNN on GPUs in the pipeline manner. PipeDream [22] introduces a pipeline approach to reduce communication overhead for synchronized training with the parameter server architecture [31]. GPipe [25] uses pipelines to address the memory bottleneck for large NNs. However, pipeline parallel training usually does not retain the exact semantics of the original model: multiple versions of parameters exist during training (similar to asynchronous training), which may lead to prolonged model convergence, or convergence to a different accuracy. *FastT* does not have this problem when used for strong scaling. On the other hand, these pipeline strategies can be complementary to *FastT*. After *FastT* obtains operation placement and execution order, it can further split a mini-batch into micro-batches and allow pipelined training in the similar fashion as proposed in Gpipe.

8 Concluding Discussions

This paper presents *FastT*, a transparent module on TensorFlow to automatically find satisfying operation splitting, device deployment and execution order for DNN models running over multiple GPUs. We carefully design the system architecture and propose efficient heuristics with theoretical performance bound. *FastT* achieves up to 63.6% speed-up as compared with pure data parallelism, and outperforms representative approaches as well in terms of per-iteration training time, strategy computation time and resource consumption, or generality. It is applicable to different types of DNN models and requires no modification of the origin ML code for developers using TensorFlow.

Looking forward, we have noticed that some new features have been published in TensorFlow which allow cycles in computation graphs, such as dynamic RNN layers. Currently, *FastT* does not handle graphs with cycles. A potential solution is to break the cycles and reorganize the graph to be a DAG. We leave this as future work. Further, we build most parts of the framework on TensorFlow's Python Client API, so *FastT* currently supports developers who use Python to build their models. We will migrate our modules to TensorFlow kernel to support more APIs.

Acknowledgement

This work was supported by Alibaba Group through Alibaba Innovative Research (AIR) Program, and grants from Hong Kong RGC under the contracts HKU 17204619 and 17208920.

APPENDIX

Proof of Lemma 1

Proof. We first set the exit operation to be the last operation in chain X, $o_{i_1} = o_{exit}$. There are three possibilities regarding $AST(o_{i_1})$:

1): $AST(o_{i_1}) \leq b_1^l$.

2): $AST(o_{i_1}) \in B$.

3): $AST(o_{i_1}) \in A$ and $AST(o_{i_1}) > b_1^l$.

For **the first possibility**, the operation o_{i_1} itself is the chain X that covers B. Now, we only consider the second and third possibilities, respectively. The idea is to extend X by one operation in each iteration to further cover some parts of B.

For **the second possibility**, there must exist such an integer $n \leq N$ that $b_n^l < AST(o_{i_1}) \leq b_n^r$. In addition, there exists a device, d_α, that is idle during $(AST(o_{i_1}) - \epsilon, AST(o_{i_1}))$ for some positive ϵ. There are again three cases in the second possibility:

▹ **Case 1:** There exists no such operation o that executes on device d_α later than $AST(o_{i_1}) - \epsilon$.

In such case, no other operation prevents o_{i_1} from being deployed at device d_α in an earlier time. Thus, we must have $EST(o_{i_1}, d_\alpha) > AST(o_{i_1})$. Let o_{i_2} represent the immediate predecessor of o_{i_1} whose data will be transmitted to o_{i_1} at $EST(o_{i_1}, d_\alpha)$ if o_{i_1} was deployed onto d_α. We have $c_{i_2,i_1} \geq EST(o_{i_1}, d_\alpha) - AFT(o_{i_2}) > AST(o_{i_1}) - AFT(o_{i_2})$. This shows that the maximal data transmission time between o_{i_1} and o_{i_2} covers the interval $(AFT(o_{i_2}), AST(o_{i_1}))$. Therefore, some parts of B are covered as we add o_{i_2} to the chain.

▹ **Case 2:** There are operations that execute on device d_α later than $AST(o_{i_1}) - \epsilon$. But the ranks for all such operations are strictly less than $rank_u(o_{i_1})$.

Because the ranks of all the later operations on d_α are strictly less than $rank_u(o_{i_1})$, indicating that at the time when o_{i_1} is scheduled, no task on d_α blocks o_{i_1} to be deployed on it. Therefore, we can always add another o_{i_2} to the chain as we do in the first case.

▹ **Case 3:** There are operations that execute on device d_α later than $AST(o_{i_1}) - \epsilon$. And there exists at least one operation with rank no less than $rank_u(o_{i_1})$.

We denote the operation executing on d_α after $AST(o_{i_1})$ - ϵ with the largest rank as o_α. If $rank_u(o_\alpha) > rank_u(o_{i_1})$, there must exist one chain from o_α to o_{exit} that completely covers the current chain X. Thus, we first assign such chain from o_α to o_{exit} as X. Since there exists some available time ahead of $AST(o_\alpha)$, there must exist one immediate predecessor of o_α whose data will be transmitted to o_α at $AST(o_\alpha)$. Similarly to case 1, we add this operation to X to further cover some parts of B.

One special case is that $rank_u(o_\alpha) = rank_u(o_{i_1})$, The decision order of o_α and o_{i_1} is uncertain. If o_α is decided first, we can substitute current X with one chain from o_α to o_{exit} and add one immediate predecessor of o_α to X as above. If o_{i_1} is decided first, this is exactly the same as case 2 and we can extend X accordingly.

For **the third possibility**, there exist an integer h that $b_h^r < AST(o_{i_1}) \leq b_{h+1}^l$ or $h = N$. We define a new set $\hat{O} = \{o | o \text{ is a predecessor of } o_{i_1} \text{ with } AST(o) > b_h^r\} \cup o_{i_1}$. And we use o_* to denote one operation in $\hat{O}$ whose any immediate predecessor o_- satisfying $AST(o) \leq b_h^r$. We add the longest chain from o_* to o_{i_1} to X first. We denote the immediate predecessor of o_* with the largest AFT as o_-. Now, there are two cases regarding the actual finish time of o_-:

▹ **Case 1:** $AFT(o_-) \geq b_h^r$

In this case, the execution time for o_- covers some part of B. Moreover, we add o_- to X in order to further extend the coverage of B.

▹ **Case 2:** $AFT(o_-) < b_h^r$

In case 2, there exists a device, d_α, that is idle during $(b_h^r - \epsilon, b_h^r)$ for some positive ϵ. We further have two subcases:

◦ **Subcase 1:** The ranks for all operations executing on d_α later than b_h^r are strictly less than $rank_u(o_-)$.

In such subcase, no task on d_α blocks o_- to be deployed on it. Then, we can perform the same procedure as in case 2 of the second possibility to add an immediate predecessor of o_- and o_- itself to X. This extension of X further shifts the coverage of B left.

◦ **Subcase 2:** At least the rank for one of the operations executing on d_α later than b_h^r are no less than $rank_u(o_-)$.

We denote the operation executing on d_α after b_h^r with the largest rank as o_α. Again, we encounter a situation similar to case 3 of the second possibility. If $rank_u(o_\alpha) > rank_u(o_{i_1})$, we use the largest chain between o_α and o_{exit} as the new chain X. And, we add the immediate predecessor of o_α whose data transmits to o_α at $AST(o_\alpha)$ to X. Now, X further covers some parts of B. If $rank_u(o_\alpha) = rank_u(o_{i_1})$, the same discussion in case 3 of the third possibility can be applied here to extend X.

For the newly added operation, there are again three possibilities as o_{i_1} and we can apply the same technique above to further extend the coverage of B using X. Eventually, we can construct a chain X that covers B completely. □

Proof of Theorem 1

Proof. Based on Lemma 1, we construct a chain $X : o_{i_M} \rightarrow o_{i_{M-1}} \rightarrow \ldots \rightarrow o_{i_1}$ covering B. ω_{idle} represents the total idle interval of all the devices. By saying X covering B, we indicates:

$$\omega_{idle} \leq |D|(\sum_{m=1}^{M} w_{i_m} + \sum_{m=1}^{M-1} c_{i_m, i_{m+1}}) \leq |D|(\sum_{m=1}^{M} w_{i_m} + C_{max})$$

By the definition of ω_{opt}, we obtain:

$$\sum_{m=1}^{M} w_{i_m} \leq \omega_{opt} \text{ and } \sum_{o_i \in O} w_i \leq |D|\omega_{opt}$$

Noting that overall execution time of all operations plus idle time of all devices is $|D|$ times the end-to-end processing time, we conclude that:

$$\begin{aligned} \omega_{DPOS} &= \frac{1}{|D|}(\omega_{idle} + \sum_{o_i \in O} w_i) \\ &\leq \frac{1}{|D|}(|D|(\omega_{opt} + C_{max}) + |D|\omega_{opt}) = 2\omega_{opt} + C_{max} \end{aligned}$$

□

References

[1] 2015. LeNet-5, convolutional neural networks. "http://yann.lecun.com/exdb/lenet".

[2] 2016. A New Lightweight, Modular, and Scalable Deep Learning Framework. "https://caffe2.ai".

[3] 2016. Tensorflow slim. "https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim".

[4] 2017. Tensorflow in-graph implementation. "https://github.com/tensorflow/examples/blob/master/community/en/docs/deploy/distributed.md".

[5] 2017. Tensorflow RunMetadata. "https://www.tensorflow.org/api_docs/python/tf/RunMetadata".

[6] 2017. Tensors and Dynamic neural networks in Python with strong GPU acceleration. "https://pytorch.org".

[7] 2018. Tensorflow Mesh. https://github.com/tensorflow/mesh.

[8] 2019. GNMT v2 For TensorFlow. "https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/Translation/GNMT".

[9] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*.

[10] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2018. Placeto: Efficient Progressive Device Placement Optimization. In *NIPS Machine Learning for Systems Workshop*.

[11] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[12] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. 2018. Efficient and Robust Parallel DNN Training through Model Parallelism on Multi-GPU Platform. *arXiv preprint arXiv:1809.02839* (2018).

[13] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[14] Xie Chen, Adam Eversole, Gang Li, Dong Yu, and Frank Seide. 2012. Pipelined back-propagation for context-dependent deep neural networks. In *Thirteenth Annual Conference of the International Speech Communication Association*.

[15] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2019. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *ACM SIGOPS Operating Systems Review* (2019).

[16] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

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

[18] Yuanxiang Gao, Li Chen, and Baochun Li. 2018. Post: Device placement with cross-entropy minimization and proximal policy optimization. In *Advances in Neural Information Processing Systems*. 9971–9980.

[19] Yuanxiang Gao, Li Chen, and Baochun Li. 2018. Spotlight: Optimizing device placement for training deep neural networks. In *International Conference on Machine Learning*.

[20] Apostolos Gerasoulis and Tao Yang. 1992. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *J. Parallel and Distrib. Comput.* (1992).

[21] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A {GPU} Cluster Manager for Distributed Deep Learning. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 485–500.

[22] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377* (2018).

[23] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. 2018. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. *arXiv preprint arXiv:1803.03288* (2018).

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

[25] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, and Zhifeng Chen. 2018. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965* (2018).

[26] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks.

In *International Conference on Machine Learning*. 2279–2288.

[27] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358* (2018).

[28] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

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

[30] Mathias Lecuyer, Joshua Lockerman, Lamont Nelson, Siddhartha Sen, Amit Sharma, and Aleksandrs Slivkins. 2017. Harvesting randomness to optimize distributed systems. In *In Proceedings of the 16th ACM Workshop on Hot Topics in Networks*.

[31] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *In proceedings of 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.

[32] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org.

[33] Khanh Nguyen, Hal Daumé III, and Jordan Boyd-Graber. 2017. Reinforcement learning for bandit neural machine translation with simulated human feedback. *arXiv preprint arXiv:1707.07402* (2017).

[34] Saptadeep Pal, Eiman Ebrahimi, Arslan Zulfiqar, Yaosheng Fu, Victor Zhang, Szymon Migacz, David Nellans, and Puneet Gupta. 2019. Optimizing Multi-GPU Parallelization Strategies for Deep Learning Training. *arXiv preprint arXiv:1907.13257* (2019).

[35] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. 2019. REGAL: Transfer Learning For Fast Optimization of Computation Graphs. *arXiv preprint arXiv:1905.02494* (2019).

[36] Jay H Park, Sunghwan Kim, Jinwon Lee, Myeongjae Jeon, and Sam H Noh. 2019. Accelerated Training for CNN Distributed Deep Learning through Automatic Resource-Aware Layer Placement. *arXiv preprint arXiv:1901.05803* (2019).

[37] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 3.

[38] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. 2018. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*.

[39] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[40] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *In Proceedings of the IEEE conference on computer vision and pattern recognition*.

[41] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems* (2002).

[42] Jeffrey D. Ullman. 1975. NP-complete scheduling problems. *Journal of Computer and System sciences* 10, 3 (1975), 384–393.

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

[44] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 26.

[45] Xiaorui Wu, Hong Xu, Bo Li, and Yongqiang Xiong. 2018. Stanza: Distributed Deep Learning with Small Communication Footprint. *arXiv preprint arXiv:1812.10624* (2018).

[46] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[47] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

[48] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, et al. 2019. GDP: Generalized Device Placement for Dataflow Graphs. *arXiv preprint arXiv:1910.01578* (2019).