This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2023.3243261

Expediting Distributed DNN Training with Device Topology-Aware Graph Deployment

Shiwei Zhang, Xiaodong Yi, Lansong Diao, Chuan Wu, Siyu Wang, and Wei Lin

Abstract—This paper presents TAG, an automatic system to derive optimized DNN training graph and its deployment onto any device topology, for expedited training in device- and topology- heterogeneous ML clusters. We novelly combine both the DNN computation graph and the device topology graph as input to a graph neural network (GNN), and join the GNN with a search-based method to quickly identify optimized distributed training strategies. To reduce communication in a heterogeneous cluster, we further explore a lossless gradient compression technique and solve a combinatorial optimization problem to automatically apply the technique for training time minimization. We evaluate TAG with various representative DNN models and device topologies, showing that it can achieve up to 4.56x training speed-up as compared to existing schemes. TAG can produce efficient deployment strategies for both unseen DNN models and unseen device topologies, without heavy fine-tuning.

Index Terms—Distributed Systems, Machine Learning

1 Introduction

Deep learning (DL) has powered a wide range of applications in various areas including computer vision [1], [2], natural language processing [3], [4], recommendation systems [5], etc. Recent deep neural network (DNN) models feature a large number of parameters (e.g. BERT [6] with more than 340M parameters) to achieve superior performance [3], [6]. Large-scale distributed training using tens or hundreds of GPUs on a cluster of machines has been the norm for training these models.

State-of-the-art distributed training largely exploits a homogeneous cluster, e.g., training Bert using 8 NVIDIA V100 GPUs [7]. Nonetheless, modern AI clouds often host a number of server types equipped with different devices (e.g., A100, V100 and P100 GPUs). Only allocating the same type of GPUs/machines to each training job may well result in scattered idling resources, e.g., 1 V100 GPU on one machine and 2 P100 GPUs on another. Often, the scattered resources cannot be allocated to one job due to lack of efficient support for training on a heterogeneous cluster with existing DL frameworks, and hence largely remain idle. To better utilize the expensive AI infrastructure, enabling efficient distributed training over heterogeneous devices is the key.

Further, inter-connectivity and bandwidth across devices in an AI cloud often differ, due to different link types (e.g., PCIE or NVLink inside a machine), different co-location levels (e.g., machines in the same rack or not), etc. This adds onto the heterogeneity of the machine learning (ML) cluster used to train a DNN model.

Multiple related decisions are involved for deploying a DNN model onto heterogeneous, scattered resources for most expedited training: On which device of which machine shall we place each operator (operation placement [8], [9])? Shall we replicate one or a group of operators on multiple devices for data-parallel training (operation replication [10], [11])? Should we use AllReduce [12], [13] or the parameter server (PS) architecture [14], [15] for parameter synchronization among replicated operators, and should gradients produced by some operators be compressed to reduce inter-device communication (gradient compression [11], [16])? These decisions jointly form an exponentially large strategy space. Current practice often falls back to heuristics that consider one aspect of the strategy space at a time [17], [18], resulting in less efficient or even infeasible solutions.

Pioneering works on deploying DNN models onto heterogeneous computation resources adopt reinforcement learning and neural networks for finding distributed training strategies [10], [18], [19]. However, their models do not generalize to different device topologies and require training from scratch for each new resource configuration. This makes them impractical for AI clouds, where new resource configurations are made for each job. A generic method that can quickly find distributed training strategies for unseen device topologies is yet to be explored.

We present TAG[1], an automatic DNN deployment framework that efficiently produces optimized distributed training strategies for a given DNN model on heterogeneous resources. TAG exploits a heterogeneous graph neural network (GNN) [20] jointly with a search-based method to make fine-grained decisions on operation replication, placement, parameter synchronization and gradient compression.

The key contributions of TAG are summarized as follows:

▷ An automatic DNN deployment framework is proposed that produces optimized training graph with operation-level replication, for expedited training over any given device set and inter-device topology. It automatically inserts necessary operations to ensure mathematical equivalence before and after modifying the original DNN computation graph.

▷ A heterogeneous GNN is designed which takes both computation graph and device topology as input, and learns a generalizable policy to guide a Monte Carlo tree search (MCTS) [21], [22] for efficient operation placement strategies.

▷ To further reduce communication overhead, we adopt sufficient factor broadcasting (SFB) [23], a lossless gradient

1. We plan to open-source TAG.

compression technique, and formulate a graph-cut problem to automatically decide subgraph duplication and SFB's application in the training graph.

▷ Extensive experiments are conducted over representative DNN models and various device topologies. TAG achieves up-to 4.56x speed-up as compared to data parallelism using NCCL AllReduce and state-of-the-art training schemes on heterogeneous clusters. It can generate efficient deployment strategies for unseen DNN models on unseen device topologies without heavy fine-tuning.

2 Background

2.1 Distributed DNN training

DNN models implemented in modern ML frameworks, e.g., TensorFlow [24], Pytorch [25], and MXNet [26], can be represented by directed acyclic graphs (DAG), referred to as the *computation graph* of the respective DNN models. In a computation graph, each node is an operation (op) and the edges connecting the ops represent tensors. The ops are placed and executed on computation devices (e.g., GPUs). If an op that produces a tensor and the op that consumes the tensor are placed on different devices, the tensor needs to be transferred across devices.

Training a DNN is an iterative process. In each iteration, forward computation is performed on a batch of training data, followed by backward propagation that calculates the gradients and updates the model parameters. Data parallelism (DP) and model parallelism (MP) are two main paradigms for distributed training [6], [27]. With classic DP, each device holds a full copy of the model and processes a batch of data independently. MP puts different parts of the DNN model onto different devices, and intermediate tensors are passed between the devices during training.

A number of studies have explored device placement, op replication and hybrid DP/MP for DNN training. GDP [9], GO [28], PlaceTo [8] and HeteroG [10] use GNN to extract computation graph information and generate device assignment of ops. HDP [18] and Spotlight [29] use reinforcement learning (RL) to train an LSTM for placement decisions. FlexFlow [30] uses the Markov Chain Monte Carlo (MCMC) search algorithm to search for op placements, achieving hybrid parallelism. REGAL [31] uses RL and genetic algorithm (GA) to co-optimize placement and scheduling. Pesto [32] models placement and scheduling into an integer linear program (ILP) and solves it with off-the-shelf ILP solvers. None of the learning-based systems supports unseen device topologies, and retraining of the respective GNN or LSTM models is required when the device topology changes.

2.2 Shared AI infrastructure

A state-of-the-art AI cloud typically includes multiple racks of servers, equipped with a few representative types of GPUs of different generations. A common practice in production AI clouds is to allocate GPUs of the same type to one training job, ideally located on the same machine or machines close to each other [33]. Such resource allocation often leads to resource fragmentation, e.g., 1-2 unallocated GPUs scattered on different machines, causing substantial wastage of expensive AI resources. Further, training jobs requesting a large number of GPUs of the same model suffer from long queuing times [34], as they have to wait for all resources to be available.

The existing DL frameworks (TensorFlow, PyTorch, MXNet, etc.) and training strategies mostly support efficient distributed training over homogeneous clusters, and their training efficiency deteriorates substantially on heterogeneous resources (i.e., GPUs with different computation and memory capacities, varying inter-device bandwidth). A design to efficiently utilize the scattered, heterogeneous resources is of strong interest.

2.3 Sufficient Factors Broadcasting

Sufficient factor broadcasting utilizes the low-rank structures in gradient tensors to make mathematically equivalent transformation on the computation graph [23]. Sufficient factors are small tensors that can generate a gradient tensor, usually by an outer product. In SFB, sufficient factors are broadcast to op replicas instead of the full gradient tensor, potentially reducing communication time. SFB does not impose precision loss nor affect training convergence.

Chilimbi et al. [35] exploit low-rank structures in the last layers of CNN and advocate explicitly sending only the activation and error gradient vectors to the PSs. Poseidon [36] broadcasts sufficient factors for synchronizing gradients among workers. They only support `MatMul` layers, limiting their usage on recent models with new types of ops. These methods are not automatically enabled and an ML developer needs to choose where to apply these optimizations for the best efficiency.

3 Motivation and Challenges

3.1 Opportunities

Partial Data Parallelism. Existing frameworks either replicate an op on all GPUs, or place it on just one GPU in pure DP, MP, or hybrid DP/MP [8], [9], [10], [18], [19], [29]. In a given GPU cluster, it is possible to replicate some ops on a subset of nearby GPUs, but not on other GPUs, achieving a good trade-off between GPU utilization and parameter synchronization overhead [30].

Topology-Aware Automatic Device Placement. In existing designs, the decision NN needs to be retrained when the device topology changes (e.g., devices to use or the link bandwidth between devices change), because they do not take device topology as input to their NNs and the structure of their NNs may also need to be altered when device topology changes. For example, the output dimension of the GNN in HeteroG [10] is $M+4$, where M is the number of devices; when the number of devices changes, its GNN needs to be retrained with a new output dimension. Designing a strategy making framework that can handle various device topologies, as well as different DNNs, is a more general and practical solution.

Automatic Sufficient Factors Broadcasting. With the advance in computation power of GPU and TPU, communication overhead becomes increasingly significant in DNN training. It is especially important to minimize communication overhead when training using fragmented resources in shared AI clusters, which are scattered on different machines or racks. Gradient compression has been used to reduce tensor size for communication. Quantization [37] and

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2023.3243261

sparsification [38] are the most used gradient compression methods [16], but introduce precision loss. We focus on SFB, which can reduce communication time without affecting training convergence.

3.2 Challenges

A very large strategy space. Suppose that we are deploying a DNN model of n operations to a cluster of m GPUs. We can replicate each op on any subset of the m devices, forming $2^m - 1$ choices. Also, for each replicated parameter (assuming there are n_p replicated parameters), we choose between AllReduce or PS architecture for its parameter synchronization. In total $(2^m - 1)^n + 2^{n_p}$ strategies are to be explored.

RL methods often suffer from overfitting. Reinforcement learning (RL) has shown success in exploring large search spaces [31], [39]. However, RL-based methods tend to overfit the training data and may lead to poor generalization performance [28], [40], [41]. Conventional regularization methods such as data augmentation [40] hardly help because the models and device topologies used in practice may differ drastically from those used in training.

SFB is not always beneficial. Communication cost of SFB depends on the number of replicas and size of sufficient factors (which is highly related to the batch size). The communication-minimization choice among using SFB or a gradient synchornization method (AllReduce or PS) needs to be examined for each gradient.

3.3 Solutions

To exploit the above opportunities and address the challenges, we propose TAG, an automatic DNN deployment framework, with the following key designs.

Interactive strategy refinement with runtime feedback. Existing schemes like GDP [9] and HeteroG [10] generate strategies in a one-shot way: their models directly output the strategy for the whole graph. Though the results are impressive, it is hard to interpret and reason about the decision process. On the contrary, human developers often optimize distributed training strategies iteratively and interactively. A developer may first run a simple strategy and examine the execution trace to find out the bottleneck of this strategy and improve it accordingly. For example, if a strategy results in low utilization and long idle time on one GPU, it may be worth considering putting more ops onto this GPU. Sec.5 in [15] gives a concrete example of such practice. Based on this observation, we design an automatic, interactive strategy-making process in TAG. Instead of taking only the raw features of a model as input and directly generating a strategy, TAG repeatedly simulates the execution of a strategy and collects the corresponding simulated execution trace, and then tries to generate a better strategy based on the runtime feedback for the existing strategy. The interactive strategy exploration process also helps TAG produce feasible (e.g. not causing OOM errors) strategies for unseen models. With systems that generate strategies in one-shot, if the generated strategy causes OOM, user intervention is required to tune the parameters and retry. With interactive strategy exploitation, TAG automatically tries to reduce the memory usage by more aggressive model parallelism when OOM errors are encountered, until a feasible solution is found.

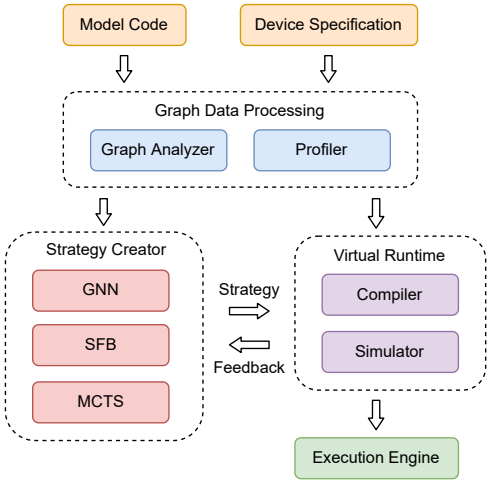


Figure 1: Workflow of TAG.

Combining learning and searching. Learning-based methods require that datasets used for training should follow the same distribution as the real use cases, or overfitting of the training data may occur [40], [41]. However, it is very hard to collect a dataset that can represent the distribution of "DNN models". For example, PlaceTo [8] is trained on only 3 models, GDP [9] is learned on 11 models, and HeteroG [10] is trained on 8 models. As new DNN models emerge, a new model can differ drastically from those in the small training datasets. To mitigate this problem, we argue that the learning-based methods should be paired with a search method that can robustly handle any unseen models. In TAG, we combine RL and MCTS, where the RL agent uses a GNN to produce prior probabilities to guide the MCTS process.

Unified representation of computation graph and device topology. Recent works have adopted GNN to produce distributed training strategies for different computation graphs [8], [9], [10]. Parameter dimensions of GNNs are not dependent on the number of nodes in a graph, allowing them to generalize to graphs of different sizes. However, only the computation graph is encoded in the input to the GNN in existing works, and their designs cannot adapt to unseen device topologies without retraining. To enable TAG to generalize to different device topologies, we use a heterogeneous graph as input, that contains both computation nodes and device nodes. This allows us to encode all inputs in a unified graph and learn a heterogeneous GNN that can generalize to both unseen DNN models and device topologies.

Formulating SFB as an optimization problem. SFB deals with gradients of the parameters, whose only consumer is the `ApplyGradient` op. Whether or not to apply SFB for a gradient is a local decision that is independent of the strategies of other ops. We formulate the SFB decision for each gradient as a mixed-integer optimization problem that can be efficiently solved using off-the-shelf solvers. The scale of the optimization problem remains small even for large graphs because it only focus on the subgraph around a gradient.

4 System Design

The workflow of TAG is given in Fig. 1. A user first builds a single-GPU DNN computation graph using a DL platform API, such as that provided in TensorFlow. The computation

graph is fed to TAG along with the device specification, describing the available GPUs allocated for training this DNN. Though we focus on distributed training in this paper, TAG supports both training and inference tasks. For training jobs, the input computation graph should contain both forward and backward ops (e.g., generated by TensorFlow's automatic differentiation engine). For inference tasks, the input computation graph only contains the ops in the forward pass.

TAG first extracts necessary features from the computation graph and device specification. A profiler collects computation time of each op on each type of GPUs and the collective communication performance among the GPUs. With these data, TAG iteratively invokes the strategy creator and the virtual runtime to find good deployment strategies. TAG does not rely on any domain knowledge nor distinguish specific operators except for some special ones such as `Placeholder`. It is hence generic and can support user-defined operators.

The strategy creator identifies a deployment strategy for a given DNN model. In the virtual runtime module, the compiler modifies the computation graph accordingly, inserting necessary auxiliary ops that ensure the modified graph is equivalent to the original graph. The simulator simulates the execution of the modified graph and estimates the training time as feedback for the strategy creator to produce new and potentially better strategies.

4.1 Graph Data Processing

4.1.1 Graph Analyzer

The graph analyzer builds an internal representation of the original computation graph, that is independent of the API used.

Simplifying the graph. Unnecessary nodes are removed including `identity`, `NoOp` and the dangling ops that are not connected to the main optimization ops (e.g., `GradientDescent`, `Adam`). This reduces the graph size without changing the semantics.

Annotating tensor split and concatenation methods. If two producer-consumer ops are replicated on different numbers of devices, split and concatenation ops are added: replicated tensors are concatenated before being sent to an op that is not replicated, and a full tensor can be split with different parts sent to different replicas of a replicated op. However, only some ops can be split and concatenated while preserving mathematical equivalence. The graph analyzer annotates each op with its splittability that will later be used by the compiler. Specifically, the graph analyzer marks every op into one of the following categories.

• *Splittable with concatenation.* Ops in this category can accept input tensors that are split in the batch dimension. When the input is split, the output of such an op is also split and can be concatenated in the batch dimension to recover the full tensor. Such ops include element-wise ops (e.g., `AddN`), batched `Conv2D` and `MaxPool2D`, etc.

• *Splittable with element-wise summation.* Ops in this category can also accept split input tensors, but the output needs to be summed, instead of being concatenated, to recover the full output tensor. This category typically includes ops that produce gradients, e.g., `Conv2DBackpropFilter`.

• *Others.* These ops do not accept split tensors. If the producer op is replicated, input tensors must be aggregated to recover the full tensor, before being consumed by such an op. Specifically, TAG marks `ApplyGradient` ops into this category, so that gradients are automatically aggregated before being applied to parameters.

Grouping ops. The number of ops varies a lot across DNN models. For example, the VGG model implemented in TensorFlow Slim [42] has 1169 nodes, while a typical implementation of BERT-Large has 26601 nodes. To efficiently produce strategies for models of different sizes, the graph analyzer employs METIS [43] to group some tightly coupled ops together. We use METIS to partition the computation graph to no more than 60 groups by minimizing the tensor sizes on the cut edges, while keeping the total computation time of each partition balanced with a balance factor of 2. Since the replication or placement strategies of different op groups may differ, additional communication is needed to aggregate and distribute the tensors on the op group boundaries. METIS minimizes the size of these tensors, and hence minimizes this communication overhead. Each op group is regarded as a single node in the graph passed to the strategy creator. A larger number of op groups allow more fine-grained strategies, at the cost of lengthened searching time. We find that 60 groups achieve a good trade-off in our experiments.

4.1.2 Profiler

To measure computation time of each op, the profiler runs single-GPU model training on each type of GPUs. Since ops can be replicated, we need the computation time under different batch sizes. We profile op execution time using typical batch sizes below 60. It is shown [44] that when the batch size is large enough, computation time is almost linear with the batch size. We build a linear model to predict the computation time for larger batch sizes that are not profiled. For a large model that does not fit in a single GPU even with small batch sizes, we manually partition the model and profile each part separately.

To predict the tensor transfer time, the profiler measures performance for GRPC transfer (between pairs of devices) and NCCL AllReduce communication (among different combinations of devices). Random 32-bit floating-point-number data of different sizes are transferred (starting from 1KB and doubled until 1GB). Segmented linear regression models are built for GRPC transfer and for AllReduce communication.

4.2 Strategy Creator

The strategy creator consists of three components. *MCTS* progressively generates placement and replication strategies for each op group. During the search, a heterogeneous *GNN* is exploited to provide prior probabilities for MCTS to sample the strategy candidates, based on a graph input that joins the computation graph with the device topology graph. When MCTS produces a strategy that replicates a parameter, *SFB solver* decides if SFB can be applied to reduce communication.

Input to the strategy creator includes a computation graph and a device graph. The former contains N nodes, each representing an op group. The latter has M nodes, each denoting a group of homogeneous GPUs, i.e., a set of GPUs

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2023.3243261

of the same type with the same bandwidth between each pair. This usually maps to a machine equipped with multiple same-type GPUs.

The strategy creator produces a deployment strategy, containing op placements and the replication plan for each replicated op. The placement P is an $N \times M$ binary matrix. $P_{i,j} = 1$ if the i-th op group is placed on one or all devices in the j-th device group (depending on the replication plan). The replication plan O is a $N \times 4$ matrix where the i-th row corresponds to a one-hot encoding of 4 replication options for the i-th op group. O_i defines how to place the i-th op group on the respective device group, if the device group includes multiple devices. The 4 replication options considered in TAG are:

• **Replicate with `AllReduce`.** The op group is replicated to all devices in the device group and input tensors are evenly split in the batch dimension. If the op group produces gradients, `AllReduce` ops are used for synchronization.

• **Replicate with PS.** It is similar to replicate with `AllReduce` except for using PS to synchronize gradients. The PS is chosen among GPUs in the device group in a round-robin manner.

• **Duplicate.** The op group will be copied to all devices in the device group. Unlike replication, input tensors are broadcast to all copies, so that gradients produced on all devices are identical, eliminating the need of synchronization.

• **Model Parallelism.** The ops in the op group are divided into smaller groups using METIS and placed to different devices in the device group. It is useful to place large models that otherwise cause out-of-memory (OOM) errors.

4.2.1 Heterogeneous GNN

The GNN takes as input a heterogeneous graph, containing two types of nodes and three types of links. The first node type is computation node, each representing a group of ops; the other type is device node, representing a homogeneous group of GPUs. The three link types include: (i) links connecting two computation nodes, corresponding to tensors in the computation graph; (ii) links that connect device nodes, representing a network link or a PCI switch; (iii) links connecting computation nodes and device nodes, denoting a specific placement of the respective op group in the device group.

Input features to the GNN contain four parts: (1) Raw features of the computation graph and devices, including total computation time (averaged over measured running time on different devices) and overall parameter size of each op group, the number of GPUs in each device group, memory capacity of each GPU, bandwidth between GPUs in each device group, size of tensors connecting two op groups, and inter-group bandwidth between each pair of device groups. (2) A strategy encoding, the one-hot encoding of replication plans of all op groups, and a binary edge feature for each edge that connects an op group and a device group, indicating whether the former is placed on the latter. (3) Runtime feedback for the input strategy, including makespan of each op group's execution, average idle time between the end of an op group's execution and the start of its output tensor transfer, peak memory usage and idling percentage of each device group, and idling percentage of each link between device groups. (4) The search progress, a one-hot encoding indicating which op groups' deployment strategies have been decided and which op group's strategy will be produced next. We add fully-connected layers to transform node-related features and edge-related features to feature vectors of fixed length f before feeding them to the GNN, so that the embedding lengths remain the same through GNN convolutions. We summarize the input features in Table 1.

Table 1: GNN input features.

Type	Feature
op node	computation time
	parameter size
	replication plan
	makespan
	idle time before transfering output
	if the strategy has been decided
	if the strategy is to be produced next
device node	number of GPUs in the group
	memory capacity of each GPU
	intra-group bandwidth
	peak memory usage
	idling percentage
op-op edge	tensor size
device-device edge	inter-group bandwidth
	idling percentage
op-device edge	placement

We adopt a 4-layer GNN. Each GNN layer transforms the embeddings produced by the previous layer by $h_u^{i+1} = \text{AGG}_{v \in \mathcal{N}(v)} \gamma_{etype} \cdot \sigma(W_{i,etype}(h_v^i \circ e_{uv}) + b_{i,etype})$. Here h_u^i is the embedding of node u at the i-th layer, $\mathcal{N}(v)$ is the set of neighbors of node v in the input heterogeneous graph, and e_{uv} is the edge feature between u and v. $\circ$ is vector concatenation. $W_{i,etype}$ and $b_{i,etype}$ are parameters of the i-th layer for $etype$ and σ is a non-linear function. AGG represents the aggregation of features from neighbors. h_u^0 is initialized as the node features of node u. e It is observed that deep GNNs suffer from the over-smoothing issue [45], [46], i.e., deeper GNNs do not necessarily perform better. Our experience confirms this phenomenon and we find that 4 layers give the best results in TAG. We choose multi-head attention based aggregation introduced in graph attention networks (GAT) [47] as it assigns different weights to different neighbors, which is desirable because we believe that some of the neighbors are more important in determining the best strategy for an op group. γ_{etype} is the weight for different types of edges: γ_{etype} is set to 1 for edges connecting the same types of nodes and 0.1 for those connecting different types of nodes, so as to balance the different types of edges connecting a node. Fig. 2 shows the structure of the GNN.

Output of the GNN includes embeddings E_{op} of op groups and embeddings E_{dev} of device groups. The GNN is further connected to a thin decoder, which contains a simple `Dense` layer that takes as input a strategy slice (P_i, O_i), containing placement and replication plan of the i-th op group, and feature vector $\sum_{j=1}^{M} E_{dev}[j] P_{i,j} \circ E_{op}[i] \circ O_i$ (involving embeddings produced by the GNN). It computes a score for the strategy slice. A softmax op further produces probabilities for all possible strategy slices of the i-th op group according to their scores. The probabilities are used for guiding MCTS's exploration.

4.2.2 Monte-Carlo Tree Search

MCTS is a best-first search algorithm that balances exploitation and exploration. In our search, a vertex in the search tree

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2023.3243261

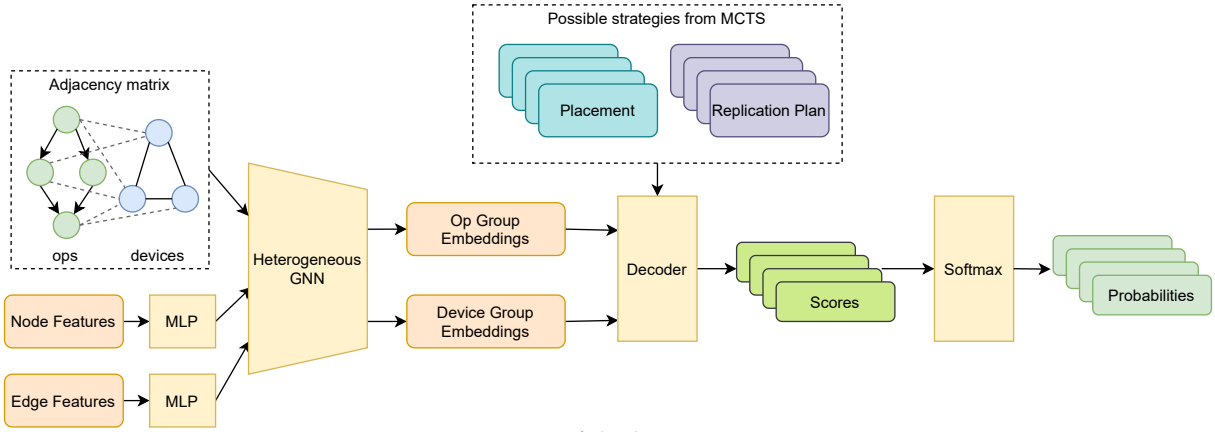


Figure 2: Structure of the heterogeneous GNN.

represents a partial deployment strategy s and an edge is an action a. The partial strategy includes incomplete placement and replication plan matrices P and O, i.e., the matrices with some rows filled, corresponding to some op groups. a is the deployment strategy to be applied to the next op group in consideration. Op groups are sorted in descending order of computation time, so that the most computation-expensive op group will be considered first. Each edge records its visit count $N(s,a)$ and a running average reward $Q(s,a)$. We use the simulator to estimate execution time of the DNN graph with the current deployment strategy and calculate the speed-up over the baseline strategy (aka DP with AllReduce-based parameter synchronization) as the reward.

MCTS starts with an empty strategy s_0, and progressively builds a search tree by repeatedly performing the following:

• *Selection*: Starting from the root of the tree, keep selecting child vertices to traverse, until we reach a leaf vertex (a vertex which has not been expanded, or with complete deployment strategies for all op groups). At each vertex s, the edge (s,a) with the highest PUCT score [22] is picked to traverse next, as defined as:

$$U(s,a) = Q(s,a) + cG(s,a)\frac{\sqrt{\sum_{a' \in \phi} N(s,a')}}{1+N(s,a)}$$

where c is a coefficient, $G(s,a)$ is the prior probability produced by the GNN, and ϕ denotes all actions in the child vertices of s. The PUCT score prioritizes the most promising strategies for exploration.

• *Expansion and Evaluation*: When reaching a leaf vertex, we evaluate the reward r of this vertex, which is the training speed-up achieved by its strategy over the baseline (DP), using the simulator.[2] If an OOM error results, the reward is set to -1. Then we expand the subtree by one level, by enumerating possible strategies for the next op group and obtain their prior probabilities $G(s,a)$ from the GNN.

• *Back-propagation*: After obtaining reward r at a leaf vertex, the running average reward $Q(s,a)$ and the visit count $N(s,a)$ along the path from the root vertex to the leaf vertex are updated, e.g., $Q(s,a) := Q(s,a) + \frac{1}{N(s,a)} \cdot r$.

Fig. 3 shows an example of the MCTS search tree. The root node is the empty strategy s_0. Each level includes the placement and replication plan for an op group. In each

2. For op groups whose deployment strategies have not been decided, we use the strategy of the most computation-expensive op group on them.

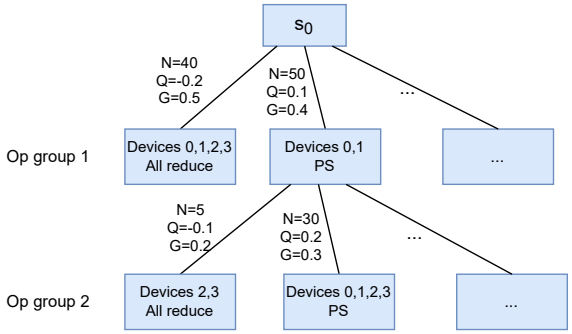


Figure 3: Example of an MCTS search tree.

iteration, the search tree is traversed from the root node to a leaf node according to the selection policy. Then it expands one level and evaluates the reward to update the Q and N on the path.

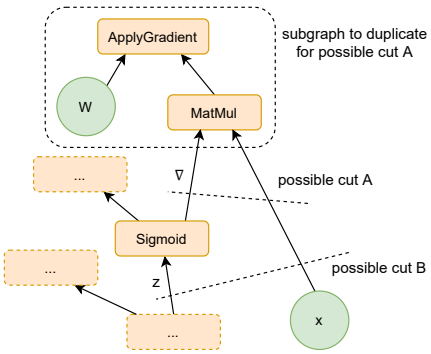
We choose MCTS over other searching methods because of the following: *First*, it allows natural integration with a RL-based method (for updating GNN model parameters), which is important for generalizability. In each step of GNN training, we randomly choose a DNN graph and a device topology. We run MCTS and collect the selection probability $\pi(s) = \text{softmax} \ln N(s)$ at vertices with at least 800 visit counts, where $N(s)$ is a vector including visit counts of all child vertices of s. Parameters θ are then updated to minimize the cross entropy between the prior probability $G_\theta(s,a)$ produced by the GNN and the selection probability $\pi(s,a)$ of the MCTS. *Second*, MCTS builds the strategy progressively. At each vertex, we evaluate the partial strategy and use the runtime feedback to help GNN make better prediction for further strategies.

4.2.3 *Sufficient Factor Broadcasting*

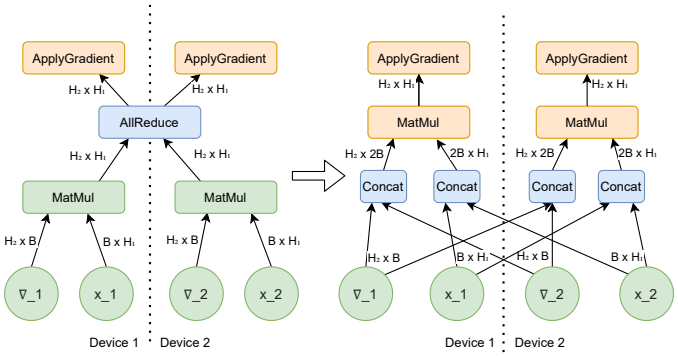
When a parameter is replicated on multiple devices (with the corresponding op), its gradients need to be synchronized across the devices. This is usually achieved by inserting an `AllReduce` or PS op. Alternatively, some gradients can be calculated from tensors of small sizes. For example, a non-full-rank gradient matrix can be represented by the product of two smaller matrices (the small matrices are the sufficient factors). Our SFB solver is designed to automatically find these mathematically equivalent replacements in the graph that can potentially reduce communication.

Fig. 4(b) shows an example of applying SFB to an `MatMul` op that can be found in the `Dense` layers in many

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2023.3243261



(a) Example cuts for SFB.



(b) Implementation of SFB for cut A in (a).

Figure 4: An example of sufficient factor broadcasting.

DNNs. When `MatMul` is replicated onto multiple devices, instead of using `AllReduce` or PS to synchronize produced gradients, the sufficient factors, ∇ and x, are broadcast to all devices, and `MatMul` ops on each device can reconstruct identical gradients. In TAG, this corresponds to choosing the "Duplicate" option for the respective op as the deployment strategy. It changes the total communication data from the gradient size, $H_2 \times H_1$, to the size of the sufficient factors, $2(H_2 \times B + B \times H_1)$, where H_1 and H_2 are the input and output feature lengths of the `Dense` layer and B is the batch size. Our SFB optimization identifies gradients where this change is beneficial.

A DNN computation graph includes optimizer ops that update parameters in the model. Consider such an op l (e.g., `ApplyGradient` in Fig. 4). One of its input op g (e.g., `MatMul` in Fig. 4) produces the gradient of the parameter that l updates. If the op group containing g is replicated by MCTS and tensor (g, l) needs to be synchronized among all replica devices, we check if SFB can be applied to reduce communication. Note that MCTS can also directly produce "Duplicate" option to enable SFB on some op group. Here we are double checking opportunities of applying SFB on op groups for which MCTS has produced a replication option. The rationale lies in that MCTS produces strategies on the op group level and the group boundaries decided by METIS are rarely the best cuts for SFB.

For every gradient tensor in a replicated op group, we solve the following optimization problem to determine a subgraph (e.g., the dashed box in Fig. 4(a)) that can be duplicated. Since sufficient factors contain all inputs used to calculate a respective gradient, they form a cut that separates the subgraph from the rest of the computation graph. For example, in Fig. 4(a), ∇ and x are a set of sufficient factors

Table 2: Notation in SFB Optimization.

E	the set of tensors inside the op group
V	the set of ops in the op group
l	an optimizer op
g	an op that produces gradient for l
D	the number of devices that have a replica of g
T_i	computation time of op i
L_{ji}	size of tensor (j, i)
τ	bottleneck bandwidth between the D devices
α_i	whether or not to duplicate op i
b_{ji}	whether tensor (j, i) is in the cut or not

of the gradient produced by `MatMul` and they form a cut; the gradient of W can be calculated by running the subgraph based only on ∇ and x. Existing studies [23], [36] only consider SFB for ops that matmul two vectors, i.e., gradients that are outer products of two vectors; our approach can identify other cases as long as such cuts can be found.

$$\min \ (D-1)\sum_{i\in V}\alpha_i T_i + D(D-1)\sum_{(j,i)\in E} b_{ji}\frac{L_{ji}}{\tau} - 2\alpha_g\frac{D-1}{D}\frac{L_{gl}}{\tau}$$

$$\text{subject to:} \quad \alpha_k \le \sum_{(k,i)\in E}\alpha_i, \quad \forall k \in V \setminus \{l\}$$
$$b_{ji} \ge \alpha_i - \alpha_j, \quad \forall (j,i) \in E$$
$$\alpha_i \in \{0,1\}, \quad \forall i \in V$$
$$b_{ji} \in \{0,1\}, \quad \forall (j,i) \in E$$

Notation is given in Table 2. α_i's are the main decisions: $\alpha_i = 1$ means changing the replication option of op i from "Replicate with AllReduce" or "Replicate with PS" to "Duplicate"; and $\alpha_i = 0$, otherwise. b_{ji} specifies if tensor (j,i) is in the cut that partitions the subgraph containing op i out.

The first term in the objective is the extra computation time due to duplication. Since we duplicate the ops in all devices where the ops' replicas are placed, each device needs to process the ops $D-1$ more times than not to duplicate. The second term minus the third term is the extra communication time incurred. For every tensor that is produced by a replicated op and consumed by a duplicated op, $D(D-1)$ transfers of the tensor are needed to broadcast it to all devices involved in the duplication. If no duplication is applied, the gradient can be synchronized with an AllReduce or PS method, and the third term formulates the communication time of using ring AllReduce (as an example, and the case of other AllReduce algorithms and PS can be formulated accordingly). The first constraint specifies that an op is included in the duplicated subgraph only if one of its consumer ops is in it, as otherwise it is unrelated to the gradient. The second constraint ensures that a tensor (j, i) is in the cut if i is duplicated and j is not. It is an integer linear program similar to the min-cut problem, but with additional node weights on one side of the cut. We use the Cbc [48] solver to solve the problem.

4.3 Virtual Runtime

The virtual runtime evaluates a strategy without actually running it on a physical cluster. It also generates the distributed training graph according to the strategy, which can

then be loaded and executed by the execution engine (e.g. TensorFlow).

4.3.1 Compiler

The compiler takes as input the DNN computation graph and a deployment strategy (P, O) found by the strategy creator. It applies the strategy and produces a modified computation graph for either the simulator or the actual execution engine, e.g., TensorFlow. The compiler automatically inserts necessary auxiliary ops to ensure the equivalence of the modified graph and the original graph regardless of the deployment strategy. It first maps the ops to devices according to the placement P, and then inserts auxiliary ops in the following cases:

• When an op is replicated but its input tensors are not, `Split` ops are inserted to split the input tensors, before feeding them to the op's replicas.

• When an op is not replicated but its input tensors are replicated, `Concat` or `AddN` ops are added to aggregate input tensors, used for parent ops marked as "Splittable with concatenation" and 'Splittable with element-wise summation", respectively.

• When both the op and input tensors are replicated but on different numbers of devices, both `Concat` and `Split` ops are inserted to adjust the number of replicas.

• When a parameter is replicated, `AllReduce` or `AddN` op is inserted depending on the replication option.

4.3.2 Simulator

The simulator implements an op scheduling algorithm similar to the default scheduler of TensorFlow. It sets up a FIFO queue for each device. When all input tensors of an op is ready, the op is inserted to the queue. Each device independently simulates the execution of the ops in its task queue using the profiled data and reports the finish time of each op.

The simulator uses reference counting to track the lifetime of tensors and estimate the peak memory usage on each device. When an op is done, its output tensors are added into memory usage of the respective device. Once all ops that use a tensor are executed, the tensor is considered de-allocated and removed from the device memory usage.

5 Implementation and Evaluation

5.1 Implementation

We implement TAG as a Python module on TensorFlow 1.14. For op grouping, we use tensor size as the edge weight and computation time as node balancing constraints when running METIS. We set a default partition number of 60 in our experiments. The profiler runs TensorFlow with tracing options to collect computation time of each op under different batch sizes. With each batch size, each model is profiled 5 times to obtain the average time.

Our heterogeneous GNN is implemented on DGL [49]. We extend the GAT [47] implementation in DGL to support heterogeneous graphs and edge features. The GNN has 55MB of parameters. We use Cbc [48] to solve the SFB optimization problem. In our experiments, we find that it can reliably solve the integer optimization problem within hundreds of milliseconds.

Table 3: Benchmark DNN models.

Model	Batch size	# of ops	Parameter size
InceptionV3 [2]	96	5312	90M
ResNet101 [1]	96	7951	169M
VGG19 [51]	96	1169	548M
Transformer [52]	480	16859	407M
BERT-Small [6]	96	5061	98M
BERT-Large [6]	16	26601	2313M

5.2 Experimental Set-up

Hardware. We conduct the experiments on two clusters. The first cluster is an on-premise cluster (*testbed*) of 7 physical machines: one is equipped with 4 NVIDIA Tesla V100 32GB GPUs; four are each equipped with 2 NVIDIA GTX 1080Ti GPUs; the other two are each equipped with 2 NVIDIA Tesla P100 GPUs. The first machine has NVLink, while PCIe is used on the other machines. The machines are connected to a 100Gbps switch. The second cluster is on a public cloud (*cloud*) with 6 machines and 32 GPUs. Two of the machines are equipped with 8 NVIDIA Tesla V100 16GB GPUs and the other 4 with 4 NVIDIA Tesla T4 GPUs. These machines are inter-connected with 10Gbps bandwidth.

Benchmarks. We experiment with 6 representative DNN models for image classification and neural language processing, as listed in Table 3. Adam [50] optimizer is used in the experiments.

GNN Training. We train the GNN in strategy creator using the 6 DNN models, the testbed device topology and randomly generated 100 device topologies as input. A random device topology is produced with a machine number in [1, 6], [1, 8] GPUs per machine of a GPU type among 3 types, intra-machine bandwidth between [64, 160] Gbps (to simulate the absence or presence of NVLink) and inter-machine bandwidth within [20, 50] Gbps (to reflect different machine locations). It takes around 2 days for GNN training to converge.

Baselines. We compare TAG with the following baselines. (1) **DP-NCCL**: data parallelism with NCCL [13] AllReduce for parameter synchronization. It is widely used for distributed training and is built-in in most DL frameworks [24], [25], [26]. We implement DP-NCCL using standard in-graph replication on TensorFlow. (2) **DP-NCCL-P**: data parallelism with batch sizes allocated to the devices being inverse propotional to their computation capacities. (3) **Horovod** [12]: a data-parallel training framework (used with TensorFlow in our experiments) that incorporates optimizations such as overlapping of AllReduce and backward computation. (4) **FlexFlow** [30]: a distributed deep learning framework that supports parallelization in the SOAP dimensions. FlexFlow assumes that the cluster is homogeneous. We implemented VGG19, ResNet101 and InceptionV3 on it using its Keras API. Due to the lack of attention layer implementation, we were not able to implement all our benchmark models on FlexFlow. We set the number of its MCMC search iterations to 100000. (5) **HDP** [18]: a heterogeneity-aware hierarchical device placement system that joinly learns grouping and device allocations with reinforcement learning. (6) **Post** [53]: a device placement system with cross-entropy minimization and proximal policy optimization. (7) **PlaceTo** [8]: a device placement system using GNN and reinforcement learning.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2023.3243261

Table 4: Deployment strategies produced by TAG.

Model	Replication			Communication	
	V100	*1080Ti*	*P100*	*PS*	*AR*
InceptionV3	4.0	8.0	0.0	100%	0%
VGG19	4.0	2.1	0.0	100%	0%
ResNet101	4.0	8.0	4.0	67%	23%
Transformer	4.0	8.0	0.0	1.7%	98.3%
Bert-Small	3.9	0.1	0.1	15.0%	85.0%
Bert-Large	0.6	0.5	0.1	3.5%	0%

(8) **GDP** [9]: a device placement system using GNN and Transformer models. (9) **Baechi** [54]: an algorithmic device placement system. We only compare with its *mSCT* algorithm as it is reported to outperform the other two algorithms proposed in the paper. (10) **HeteroG** [10]: a state-of-the-art system that supports heterogeneous clusters, which uses a GNN to make op deployment decisions. It supports a similar decision space as TAG, but only assumes one given device topology and replicates an op to all devices or put it on a single device. We train HeteroG with all the benchmark models under the device topology of our testbed.

Some of the baseline systems are not open-source and non-trivial to reimplement. We adopt the evaluation methodology used in related work [10], [54] and compare the reported improvements over expert strategies for these baselines.

5.3 Training Speed-up on Heterogeneous Clusters

We first compare the average per-iteration training time incurred by TAG and the open-source baselines on our testbed (Sec. 5.2). Fig. 5 shows that TAG outperforms the baselines across all models: 8%-456% speed-up compared with DP-NCCL, 1%-391% speed-up compared with DP-NCCL-P, 11%-381% speed-up compared with Horovod, and 4%-186% speed-up compared to HeteroG. Note that HeteroG is trained from scratch for this device topology. The best acceleration is achieved over DP-NCCL when training VGG19: VGG19 has a relatively large number of parameters and communication tends to be the bottleneck; with DP-NCCL, parameter synchronization happens after the slowest devices are ready, which slows down training substantially. Models such as ResNet101, on the other hand, have less parameters but are more computation-intensive; DP-NCCL utilizes all GPUs well and both TAG and HeteroG can hardly find much better strategies. Horovod outperforms DP-NCCL in most cases due to its highly optimized communication, but the speed-up is very limited in a heterogeneous cluster. DP-NCCL-P balances the computation workload of different cards and performs better than DP-NCCL in the heterogeneous cluster. However, the overall improvement is very limited due to two reasons. First, it has the same communication time as DP-NCCL: both methods synchronize all parameters among all devices with AllReduce. Second, different operators have different computation characteristics. The optimal batch size distribution among the devices varies for different operators. FlexFlow finds hybrid strategies that choose different replication numbers and placements for different layers. However, since FlexFlow does not consider the heterogeneity among computation devices, it puts an excessive amount of workload on slow cards and results in suboptimal performance.

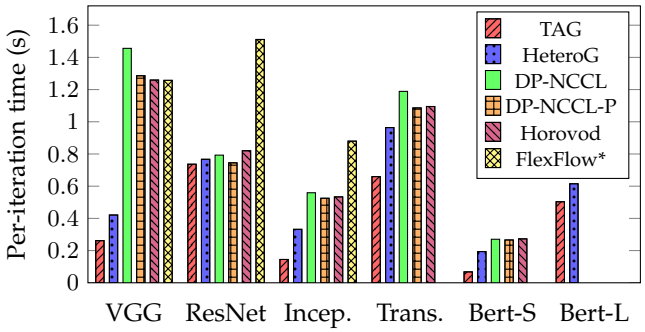


Figure 5: Per-iteration training time. DP-NCCL, DP-NCCL-P, and Horovod result in OOM with BERT-Large. *FlexFlow is implemented on Legion [55] while other systems are implemented on TensorFlow [24].

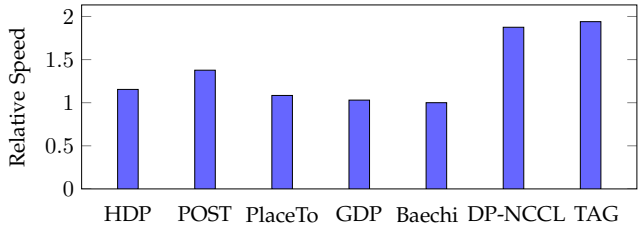


Figure 6: Training speed of InceptionV3 on homogeneous clusters. The speed is relative to the human expert strategy.

Table 4 provides details of the strategies produced by TAG, on the average number of GPUs of each type that ops are replicated onto and percentages of gradients that use PS or AllReduce for synchronization. For ResNet101, TAG replicates all ops onto all devices; for other models, the 4 P100 GPUs are rarely exploited, because benefits provided by using these devices do not cover the additional communication costs. We observe that a mixture of PS and AllReduce is used for parameter synchronization when training most models; the "duplicate" option is not selected because it is mainly effective with small batch sizes (as in the case to be presented in Sec. 5.6), and batch sizes used in this experiment are relatively large.

5.4 Training Speed-up on Homogeneous Clusters

We compare TAG with more baselines, including those not providing open-source implementation, by comparing the reported speed-up over human expert strategies following the evaluation methodology in the related work [10], [54]. For fair comparison, we conduct this experiment under a similar hardware setting as in these works, which is a homogeneous cluster with two V100 GPUs on the same machine. We use InceptionV3 as the benchmark model because it is the only common model evaluated in all these works. The same expert strategy as in existing studies [18], [53] is used for the benchmark model. As shown in Fig. 6, TAG outperforms all baselines by 3%-94%.

5.5 Effectiveness of the Runtime Feedback Features

Among the four parts of feature input to our GNN (Sec. 4.2.1), part 3 includes a number of features provided by the simulator. Other studies which use a simulator to drive GNN

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2023.3243261

Table 5: Per-iteration training time (seconds) with and without applying sufficient factor broadcasting.

Model	DP-NCCL			TAG			FlexFlow
	without SFB	*with SFB*	*Speedup*	*without SFB*	*with SFB*	*Speedup*	
InceptionV3	0.0898	0.0452	98.7%	0.0775	0.0420	84.5%	0.0631
ResNet101	0.1122	0.0854	31.4%	0.0508	0.0490	3.8%	0.0552
VGG19	0.1195	0.1192	0.3%	0.1167	0.1169	-0.2%	0.1083
Transformer	0.1199	0.0455	163.5%	0.0521	0.0451	15.5%	N/A
BERT-Small	0.0618	0.0546	13.2%	0.0597	0.0535	11.6%	N/A
BERT-Large	0.4576	0.4317	6.0%	0.4331	0.4266	1.5%	N/A

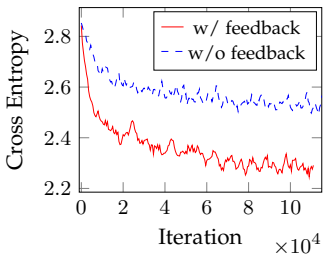


Figure 7: Loss curve of the GNN.

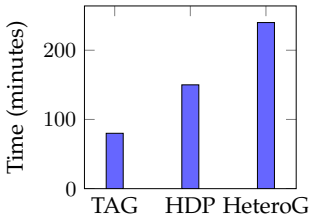


Figure 8: Overhead of generating a strategy on unseen device topologies.

Table 6: Top 5 operations that TAG chooses to duplicate.

Operation	Count
Reshape	341
MatMul	336
Transpose	89
Conv2DBackpropFilter	66
Add	26

Table 7: Average # of MCTS search iterations to obtain a better strategy than DP-NCCL.

Model	Pure MCTS	TAG
InceptionV3	66.0	9.5
ResNet101	73.4	4.6
VGG19	56.6	17.7
Transformer	145.0	121.8
Bert-Small	97.8	7.9

Table 8: Average speed-up over DP-NCCL. TAG: trained with all models. TAG–: trained with other DNNs and producing strategy for one hold-out model.

Model	Testbed		Cloud	
	TAG	*TAG–*	*TAG*	*TAG–*
InceptionV3	456.1%	456.1%	117.7%	117.7%
ResNet101	7.7%	7.1%	12.2%	10.4%
VGG19	286.2%	213.6%	43.6%	43.6%
Transformer	80.3%	80.3%	15.0%	13.4%
Bert-Small	298.4%	279.6%	84.5%	84.5%

learning (e.g., FlexFlow [30] and HeteroG [10]) largely exploit only execution time produced by the simulator, while we use multi-dimensional information estimated by the simulator as GNN input. To evaluate effect of such features, we train the GNN with and without them. Fig. 7 shows that the runtime feedback features significantly boost the learning of the GNN.

5.6 Effect of Sufficient Factor Broadcasting

To reveal the potential benefits of SFB, we conduct this experiment on two machines, each equipped with one 1080Ti GPU. We use a batch size of 4 for all models. We compare the per-iteration training time achieved with DP-NCCL and TAG, before and after enabling SFB, in Table 5. We also include FlexFlow as a reference. When applying SFB with DP-NCCL, we solve the SFB optimization to find a subgraph around each gradient, and replace the gradient's AllReduce synchronization by the "duplicate" option. SFB brings significant speed-up in training InceptionV3 and Transformer, suggesting that there are more low-rank structures in these models. As TAG also adopts other strategies to alleviate communication overhead, e.g., mixing PS and AllReduce, the total communication time with TAG is shorter, and the speed-up achieved by applying SFB in TAG is smaller as compared to the DP case.

We summarize the top 5 ops duplicated via SFB optimization across all 6 DNN models in Table 6. The count indicates the total number of times when the respective ops are duplicated via SFB optimization. TAG can identify SFB opportunities beyond `MatMul`.

5.7 Generalizability to Unseen Device Topologies

To evaluate generalizability of our GNN model, we randomly generate 100 unseen device topologies (in the same way as how device topologies used for GNN training are produced, as described in Sec. 5.2) and use the simulator to evaluate strategies produced by TAG for training a DNN (randomly selected out of the 6 models) on each unseen topology.

We collect the number of MCTS search iterations required for TAG to find a deployment strategy that achieves better training time than DP-NCCL. We also compare this number with the search iteration number required by pure MCTS without using prior probabilities from the GNN, but the probabilities from a uniform distribution. In Table 7, we see that with prior probabilities from the GNN, TAG can quickly find strategies that outperforms DP-NCCL, while Pure MCTS needs many more iterations.

5.8 Generalizability to Unseen Computation Graphs

We train TAG with 5 of the models in Table 3 and then produce strategies for the hold-out model (we vary the hold-out model among the 6 DNNs). We conduct this experiments on both the testbed and the cloud. As Table 8 shows, strategies produced for the unseen models are only marginally worse than those for models in the training set.

5.9 Overhead of TAG

We compare the overhead of TAG with two learning-based methods, HDP [18] and HeteroG [10]. As shown in Fig. 8, TAG is 87.5% faster than HDP and 2x faster than HeteroG.

Thanks to its generalizability to unseen device topologies, TAG only needs to run the MCTS search and GNN inference to generate a strategy, while HeteroG requires training from scratch. HDP constantly evaluates the strategy on real clusters during the search, which incurs a large overhead.

6 Conclusion

This paper proposes TAG, an automatic DNN deployment system that accelerates distributed training over scattered resources. TAG combines both the DNN computation graph and the device topology graph as input to a GNN, and integrates the GNN with MCTS to identify optimized deployment strategies. With automatic partial replication and sufficient factor broadcasting, TAG can better utilize heterogeneous resources and reduce communication overhead for DNN training. In our experiments, TAG achieves up to 4.56x speed-up as compared to representative existing schemes. TAG is generic and efficient in producing good strategies for both unseen device topologies and DNN models without re-training.

As a future direction, we plan to extend TAG to support pipeline parallelism, by expanding the replication plan to include a "pipeline" option. The graph compiler would need to add appropriate control dependencies in the distributed graph for ops to process different micro-batches in the same pipeline stage, to achieve efficient pipelining.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[2] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[4] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[5] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee, "Billion-scale commodity embedding for e-commerce recommendation in alibaba," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 839–848.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[7] J. Lee, W. Yoon, S. Kim, D. Kim, S. Kim, C. H. So, and J. Kang, "Biobert: a pre-trained biomedical language representation model for biomedical text mining," *Bioinformatics*, vol. 36, no. 4, pp. 1234–1240, 2020.

[8] R. Addanki, S. B. Venkatakrishnan, S. Gupta, H. Mao, and M. Alizadeh, "Placeto: Learning generalizable device placement algorithms for distributed machine learning," *arXiv preprint arXiv:1906.08879*, 2019.

[9] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. C. Ma, Q. Xu, M. Zhong, H. Liu, A. Goldie, A. Mirhoseini *et al.*, "Gdp: Generalized device placement for dataflow graphs," *arXiv preprint arXiv:1910.01578*, 2019.

[10] X. Yi, S. Zhang, Z. Luo, G. Long, L. Diao, C. Wu, Z. Zheng, J. Yang, and W. Lin, "Optimizing distributed training deployment in heterogeneous gpu clusters," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020, pp. 93–107.

[11] H. Zhang, Y. Li, Z. Deng, X. Liang, L. Carin, and E. Xing, "Autosync: Learning to synchronize for data-parallel distributed deep learning," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[12] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[13] S. Jeaugey, "Nccl 2.0," in *GPU Technology Conference (GTC)*, 2017.

[14] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 583–598.

[15] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed {DNN} training in heterogeneous gpu/cpu clusters," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 463–479.

[16] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.

[17] S. Kim, G.-I. Yu, H. Park, S. Cho, E. Jeong, H. Ha, S. Lee, J. S. Jeong, and B.-G. Chun, "Parallax: Sparsity-aware data parallel training of deep neural networks," *arXiv preprint arXiv:1808.02621*, 2018.

[18] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A hierarchical model for device placement," in *International Conference on Learning Representations*, 2018.

[19] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2017, pp. 2430–2439.

[20] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla, "Heterogeneous graph neural network," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 793–803.

[21] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*. Springer, 2006, pp. 282–293.

[22] D. Auger, A. Couetoux, and O. Teytaud, "Continuous upper confidence trees with polynomial exploration–consistency," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2013, pp. 194–209.

[23] P. Xie, J. K. Kim, Y. Zhou, Q. Ho, A. Kumar, Y. Yu, and E. Xing, "Distributed machine learning via sufficient factor broadcasting," *arXiv preprint arXiv:1511.08486*, 2015.

[24] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016.

[25] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[26] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[27] S. Pal, E. Ebrahimi, A. Zulfiqar, Y. Fu, V. Zhang, S. Migacz, D. Nellans, and P. Gupta, "Optimizing multi-gpu parallelization strategies for deep learning training," *IEEE Micro*, vol. 39, no. 5, pp. 91–101, 2019.

[28] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. Ma, Q. Xu, H. Liu, M. P. Phothilimtha, S. Wang, A. Goldie *et al.*, "Transferable graph optimizers for ml compilers," *arXiv preprint arXiv:2010.12438*, 2020.

[29] Y. Gao, L. Chen, and B. Li, "Spotlight: Optimizing device placement for training deep neural networks," in *International Conference on Machine Learning*. PMLR, 2018, pp. 1676–1684.

[30] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks." *Proceedings of Machine Learning and Systems*, vol. 1, pp. 1–13, 2019.

[31] A. Paliwal, F. Gimeno, V. Nair, Y. Li, M. Lubin, P. Kohli, and O. Vinyals, "Reinforced genetic algorithm learning for optimizing computation graphs," *arXiv preprint arXiv:1905.02494*, 2019.

[32] U. U. Hafeez, X. Sun, A. Gandhi, and Z. Liu, "Towards optimal placement and scheduling of dnn operations with pesto," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 39–51.

[33] H. Zhao, Z. Han, Z. Yang, Q. Zhang, F. Yang, L. Zhou, M. Yang, F. C. Lau, Y. Wang, Y. Xiong *et al.*, "Hived: Sharing a {GPU} cluster for deep learning with guarantees," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 515–532.

[34] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 945–960.

[35] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 571–582.

[36] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 181–193.

[37] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns," in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[38] N. Strom, "Scalable distributed dnn training using commodity gpu cloud computing," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[39] S.-C. Kao, G. Jeong, and T. Krishna, "Confuciux: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 622–636.

[40] C. Zhang, O. Vinyals, R. Munos, and S. Bengio, "A study on overfitting in deep reinforcement learning," *arXiv preprint arXiv:1804.06893*, 2018.

[41] X. Song, Y. Jiang, S. Tu, Y. Du, and B. Neyshabur, "Observational overfitting in reinforcement learning," *arXiv preprint arXiv:1912.02975*, 2019.

[42] Sergio Guadarrama, Nathan Silberman, "TensorFlow-Slim: A lightweight library for defining, training and evaluating complex models in tensorflow," https://github.com/google-research/tf-slim, 2016.

[43] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[44] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive all-reduce scheduling for expediting distributed dnn training," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020.

[45] D. Chen, Y. Lin, W. Li, P. Li, J. Zhou, and X. Sun, "Measuring and relieving the over-smoothing problem for graph neural networks from the topological view," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, 2020, pp. 3438–3445.

[46] C. Cai and Y. Wang, "A note on over-smoothing for graph neural networks," *arXiv preprint arXiv:2006.13318*, 2020.

[47] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[48] J. Forrest, T. Ralphs, S. Vigerske, L. Hafer, B. Kristjansson, J. Fasano, E. Straver, M. Lubin, H. Santos, R. Lougee *et al.*, "coin-or/cbc: Version 2.9. 9," *URL http://dx.doi.org/10.5281/zenodo*, vol. 1317566, 2018.

[49] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[50] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[51] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[52] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.

[53] Y. Gao, L. Chen, and B. Li, "Post: Device placement with cross-entropy minimization and proximal policy optimization," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[54] B. Jeon, L. Cai, P. Srivastava, J. Jiang, X. Ke, Y. Meng, C. Xie, and I. Gupta, "Baechi: fast device placement of machine learning graphs," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 416–430.

[55] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

Shiwei Zhang received the BEng degree from the Department of Computer Science and Technology, Harbin Institute of Technology, China, in 2017. He has been a PhD student at the Department of Computer Science, The University of Hong Kong, since July 2020. His research interest is on machine learning systems.

Xiaodong Yi received the BEng degree from the Department of Computer Science and Technology, Huazhong University of Science and Technology, China, in 2017, and the PhD degree from the Department of Computer Science, The University of Hong Kong, in 2021. His research interests include network function virtualization and machine learning systems.

Lansong Diao received PhD degree from the Department of Computer Science, Beijing Institute of Technology, China, in 2003. He had worked in EDA industry for more than 10 years. Currently, he is a Staff Engineer in Alibaba Group. His current interests include compiler and machine learning systems.

Chuan Wu (Senior Member, IEEE) received the PhD degree from the Department of Electrical and Computer Engineering, University of Toronto, Canada, in 2008. Since September 2008, she has been with the Department of Computer Science, University of Hong Kong, where she is currently a professor. Her current research interests include the areas of cloud computing, distributed machine learning systems, network function virtualization, and intelligent elderly care technologies.

Siyu Wang received the Master degree in the Department of Software Engineering, Beijing Jiaotong University, China, in 2015. Since July 2015, he has been working as an algorithm engineer for developing and optimizing deep learning systems of PAI platform in Department of Computing Platform, Alibaba Cloud. His current research interests include large-scale distributed machine learning systems and AI compiliers.

Wei Lin is currently the senior director of PAI & chief architect of big-data computation platform at Alibaba. He has more than 15 years of experience specializing in backend/infrastructure, distributed system development, storage and a large scale computation system include batch, streaming and machine learning. He has published many papers in top computer system conferences, such as NSDI, SoCC, and OSDI.