Optimizing Distributed Training Deployment in Heterogeneous GPU Clusters

Xiaodong Yi[1], Shiwei Zhang[1], Ziyue Luo[1], Guoping Long[2], Lansong Diao[2], Chuan Wu[1], Zhen Zheng[2], Jun Yang[2], Wei Lin[2]
{xdyi,swzhang,zyluo,cwu}@cs.hku.hk,{guopinglong.lgp,lansong.dls,james.zz,muzhuo.yj,weilin.lw}@alibaba-inc.com
The University of Hong Kong[1], Alibaba[2]

ABSTRACT

This paper proposes *HeteroG*, an automatic module to accelerate deep neural network training in heterogeneous GPU clusters. To train a deep learning model with large amounts of data, distributed training using data or model parallelism has been widely adopted, mostly over homogeneous devices (GPUs, network bandwidth). Heterogeneous training environments may often exist in shared clusters with GPUs of different models purchased in different batches and network connections of different bandwidth availability (e.g., due to contention). Classic data parallelism does not work well in a heterogeneous cluster, while model-parallel training is hard to plan. *HeteroG* enables highly-efficient distributed training over heterogeneous devices, by automatically converting a single-GPU training model to a distributed one according to the deep learning graph and available resources. *HeteroG* embraces operation-level hybrid parallelism, communication architecture selection and execution scheduling, based on a carefully designed strategy framework exploiting both GNN-based learning and combinatorial optimization. We compare *HeteroG* with existing parallelism schemes and show that it achieves up-to 222% training speed-up. *HeteroG* also enables efficient training of large models over a set of heterogeneous devices where simple parallelism is infeasible.

CCS CONCEPTS

• **Computer systems organization → Neural networks**; **Heterogeneous (hybrid) systems**;

KEYWORDS

Distributed training, heterogeneous environment, deep learning

1 INTRODUCTION

Deep Learning (DL) models have become increasingly complicated and large over the past years. Training of a deep neural network (DNN) is extremely time consuming. Parallelizing training using multiple workers in a distributed environment is adopted with current machine learning (ML) frameworks [1, 5, 44].

Two most common parallelization strategies are data parallelism (DP) and model parallelism (MP) [9, 40, 59]. With data parallelism, a replica of the entire neural network is placed on each device (e.g., a GPU card); each device processes a subset of the training data and synchronizes model parameter updates among different replicas. For models with large parameter sizes that cannot be fit entirely into a single device's memory, model-parallel training is adopted by assigning disjoint partitions of the DNN to different devices; no gradient aggregation is needed, but intermediate activations should be transferred across devices. Performance of model parallelism highly depends on the model-to-device assignment decisions made by ML developers.

To optimize distributed training, Krizhevsky et al. [30] and Wu et al. [60] manually optimize parallelism based on human experts' domain knowledge. Some automated frameworks [13, 39] were proposed for finding efficient model parallelism strategies. GDP [64] and Placeto [2] use Graph Neural networks (GNN) to learn operation-to-device assignment strategies. Parallax [28] utilizes hybrid PS and AllReduce communication methods in data-parallel training. All of them focus on training over homogeneous devices.

Instead, we focus on DNN training expedition in heterogeneous environments. In shared ML clusters containing GPUs of different models and many DL jobs, a new-arrival training job often faces the following situation: GPUs of its desired type are not available at its required number, while there are available GPUs of other models. With standard data parallelism, the job may have to wait for its required number of GPUs of the same model become available, or make do with the fewer number of GPUs available. The job cannot exploit available GPUs of different models due to the poor performance of data-parallel training over heterogeneous devices: the processing speed is imbalanced over different devices, and the devices and communication channels (network links across servers and internal links among multiple GPUs within a single server) are less efficiently used with synchronous training (due to waiting).

In data-parallel training, parameter server (PS) architecture [31] and AllReduce methods [36, 45] are widely used for parameter synchronization. In homogeneous environments, AllReduce usually performs better than PS [28, 31] by fully utilizing the links among all devices; in a PS architecture, the links to parameter servers may become the bottlenecks. In a heterogeneous environment, a single PS or AllReduce operation at the end of each training iteration for aggregating all parameter updates may be less efficient: parameter synchronization (aka communication) now takes longer time due to imbalanced computation speeds among devices, and low utilization of the communication channel results.

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.

CoNEXT '20, December 1–4, 2020, Barcelona, Spain
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7948-9/20/12…$15.00
https://doi.org/10.1145/3386367.3432728

We advocate fine-grained, hybrid parallelism and parameter synchronization methods among operations in a DNN model, for training acceleration in heterogeneous ML clusters. We propose *HeteroG*, an automated module that converts a single-GPU training model to a distributed one, achieving optimized training speeds. *HeteroG* generates detailed parallelism strategy, device placement, gradient communication method, and execution order for each operation in a DNN model. A novel strategy framework is designed incorporating a GNN for deciding operation parallelism, placements and communication schemes, and a combinatorial optimization problem to generate execution order of operations. Main contributions of this paper are summarized as follows:

▹ We propose an automated module to generate hybrid, operation-level parallelism schemes for expedited distributed DNN training in heterogeneous environments.

▹ We design a novel strategy framework including a GNN-based policy network and a combinatorial optimization problem, with synergy to comprehensively produce the large set of strategies enabling highly-efficient distributed training.

▹ To provide generality, a carefully designed GNN is used to learn structural information from different DNN graphs, and produce good deployment strategies for a broad range of DNN models. It decides the replication number of each operation and the device placement of these replicas to fully utilize different devices and communication channels. For replicas where gradient aggregation is needed, it also decides the communication methods (PS or AllReduce), enabling different communication modes for different gradient aggregation operations in a DNN model.

▹ An efficient heuristic is designed to solve the combinatorial optimization problem on operation execution ordering. It ensures high resource utilization (GPU devices, network links) and maximal computation-communication overlap, with proven performance bound to optimal schedule.

▹ *HeteroG* is implemented as a python module in Tensorflow. Developers only need to implement single-GPU models and invoke *HeteroG*'s simple API. *HeteroG* automatically generates a distributed training model with the strategies it finds, and deploys it in the heterogeneous cluster.

▹ We carry out extensive experiments in a heterogeneous cluster. *HeteroG* is carefully compared with existing parallelism schemes: it achieves up-to 222% training speed-up as compared to data parallelism and existing hybrid parallelism designs; it can enable efficient training of large models over heterogeneous devices where simple parallelism is infeasible. We observe that a fine-grained hybrid of parallelism strategies and gradient aggregation methods for different operations, as well as variable replica numbers across heterogeneous devices, contribute to the good performance of *HeteroG*, based on very efficient utilization of available computation and communication resources.

2 BACKGROUND AND MOTIVATION

2.1 DNN Training and Parallelism

Training a DNN is an iterative process that uses a large number of samples to tune model parameters for minimizing a loss function. In current training frameworks [1, 5, 44], different kinds of computation are implemented by different operations (such as Conv2D, MatMul), and input and output of these operations are called *tensors* (e.g., gradients, activations). The computing process can typically be represented by a DAG (Directed Acyclic Graph), whose nodes are operations and edges represent tensors.

Forward and Backward Computation. In each training iteration, one batch of samples is fed into the DNN model. Operations in forward propagation (FP) takes output of precedent operations as input and generates output based on parameters. A loss is produced based on outputs at the end of FP. After FP, gradients of model parameters are computed from back to front, i.e., backward propagation (BP). The gradients are then applied to the parameters using some optimization algorithm, e.g., Stochastic Gradient Descent (SGD).

Model Parallelism (MP). Operations in the DNN model are placed on different devices [9, 40, 59]. Each device maintains part of the parameters of the model.

Data Parallelism (DP). The dataset is partitioned into mini-batches for training at each device. Each device maintains a replica of DNN model and carries out FP and BP, and gradients from different devices need to be aggregated before applied to update parameters.

PS and AllReduce. They are two popular architectures for parameter synchronization in data-parallel training [31, 36, 45]. In a PS architecture, parameters are stored in centralized parameter servers; each worker computes its gradients based on its local dataset and parameters, pushes the gradients to PSs and pulls updated global parameters from PSs. In an AllReduce architecture, each worker computes gradients and aggregates gradients from other workers for parameter updates using an AllReduce algorithm [36, 45].

Communication in MP and DP. With model parallelism, when two adjacent operations are placed on different devices, the output of precedent operation needs to be transferred to successor operation; if the two devices are in different physical servers, network communication is involved. With data parallelism, communication occurs during gradient aggregation/parameter synchronization.

2.2 Potential Training Expedition Methods in Heterogeneous Clusters

PS could be better than AllReduce. Fig. 1 shows that a single AllReduce architecture for data-parallel training may perform well in a homogeneous cluster (GPU0, GPU1, GPU2 have the same computation power), but not in a heterogeneous environment (GPU0 is slower than GPU1 and GPU2 with computation power ratio of 1:2:2). Three adjacent operations in BP are considered, where GA represents gradient aggregation (following each BP operation). In case of imbalanced computation power of GPUs, the communication channel is not fully utilized, gradient synchronization takes longer time, and the training time is prolonged.

In the heterogeneous setting, we can use the PS architecture for parameter synchronization and let the slowest GPU run both a worker and the PS functionalities. In this way, as shown in Fig. 2(a), communication for synchronizing parameters with the slowest worker is eliminated, and training is expedited. Note that in case of PS-based parameter synchronization, GA operation includes parameter push and pull to/from the PS; in this example, the GA

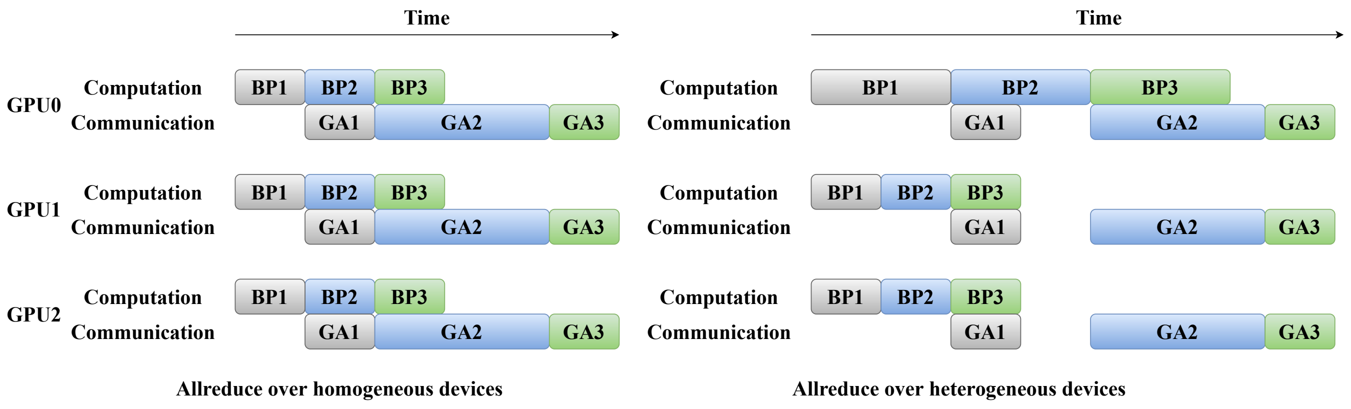
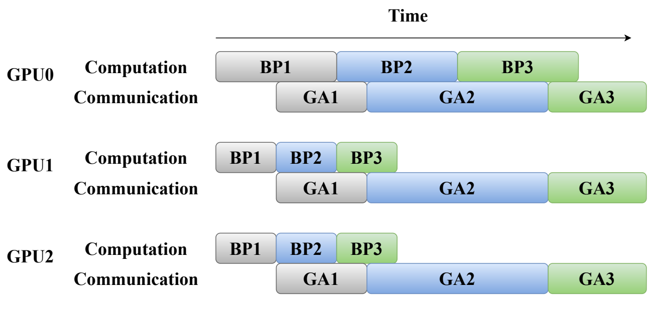
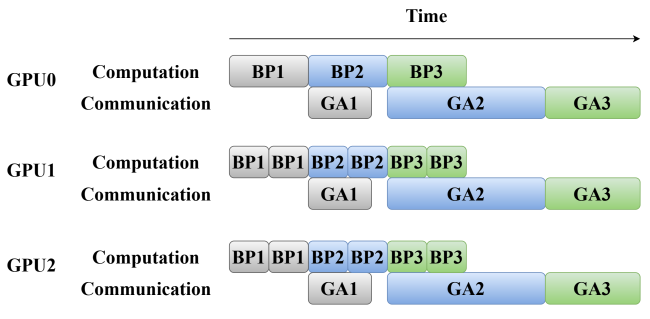

Time
GPU0
GPU1
GPU2
Computation
Communication
BP1
BP2
BP3
GA1
GA2
GA3
Allreduce over homogeneous devices
Allreduce over heterogeneous devices

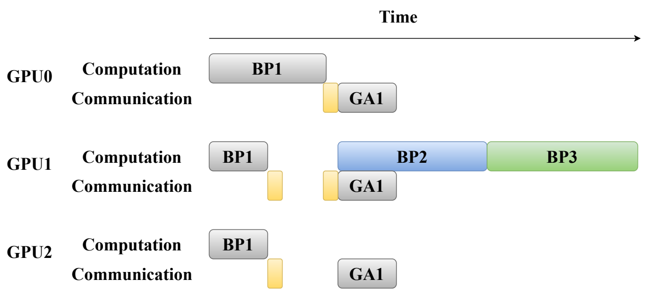
Figure 1: Prolonged training time in a heterogeneous cluster using AllReduce.



(a) GPU0 as both a worker and the PS: gradients from GPU1 and GPU2 pushed once respective BP is done.



(b) Place more operation replicas on GPU1 and GPU2.



(c) Place BP2 and BP3 in GPU1 only.

Figure 2: Potential training expedition approaches in a heterogeneous cluster.

operation at GPU0 (serving as the PS) starts when gradients are received from other devices. In a PS architecture, each worker can independently send their gradients to the PS. GA1 happens at GPU1 and GPU2 once they have finished their respective BP1, when their gradients are sent to GPU0; GA1 at GPU0 indicates receipt of these gradients from GPU1 and GPU2 (since GPU0 serves as the PS), which can start when the gradients are received and does not need to wait for the completion of BP1 in GPU0 itself.

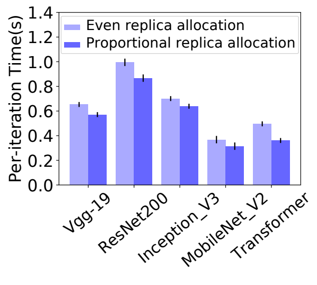
Placing more replicas to faster devices. Balancing workload among different devices can potentially lead to better utilization of computation power and communication channel. We can place more operation replicas in faster devices to achieve the balance. An operation can be replicated by dividing the input along the batch size dimension, i.e., each replica processes an even partition of the origin operation's input and its execution time is shorter than the original operation's. In Fig. 2(a), 3 replicas of each BP operation are processed on 3 GPUs; in Fig. 2(b), we make 5 replicas of each BP operation, and place a number of replicas in 3 GPUs in proportion to their computation power. In this way, we can still use AllReduce for gradient aggregation, as the GA operations are largely synchronized without long waiting time, like in a homogeneous environment.

Using MP to eliminate gradient communication. With DP, communication occurs due to gradient aggregation among multiple replicas. We can place some operations on a single device without replication (model parallelism), to reduce some communication overhead. In Fig. 2(c), BP2 and BP3 are only placed on GPU1, such that parameters in these operations are only maintained on GPU1 and no gradient aggregation (of these parameters) is needed from other devices. The small yellow rectangle denotes activation transfer time to send/receive output of BP1 from other devices to GPU1. AllReduce is used for gradient synchronization among replicas of BP1 in this example.

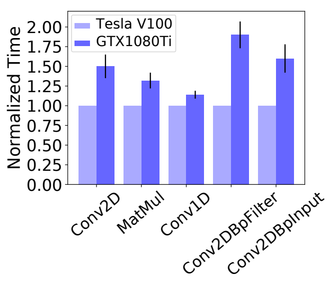
2.3 Challenges

Exploring the above opportunities comes with challenges.

PS may not be the one-for-all communication architecture in a heterogeneous cluster. In PS architecture for gradient aggregation, the links to parameter servers may become bottlenecks.



(a) Execution time with even and proportional replica allocation.



(b) Normalized average execution time of representative operations.

Figure 3: Performance of proportional distribution of whole-model replicas.

Hybrid communication methods could provide a satisfying solution: use the PS architecture for aggregating gradients of operations where link bandwidth is not the bottleneck, while exploiting AllReduce for operations whose replicas' computation is relatively balanced. However, it is difficult to judge which conditions the operations satisfy, which is closely related to placement of their replicas.

Proportional distribution of whole-model replicas may not be sufficient. We train VGG19 [49], ResNet [19], Inception-v3 [50], MobileNet_v2 [46] and Transformer [52] models respectively using DP on 4 GPUs (two Tesla V100 GPUs and two GTX 1080Ti GPUs), and compare result per-iteration training time of placing one model replica on each GPU vs. placing two model replicas on each Tesla V100 GPU and one replica on eachGTX 1080Ti GPU (computation power of the two types of GPU is roughly at the ratio of 2:1). Fig. 3(a) shows that the speed-up with proportional workload allocation is small, about 9 $\sim$ 27%. We further investigate execution time of some representative operations in VGG19 and Transformer, when each is run on a Tesla V100 GPU and a GTX 1080Ti GPU, respectively. Fig. 3(b) shows normalized operation execution time by dividing the real time by that of running on the V100 GPU. The average speed-up when using the V100 GPU varies significantly from 1.1 to 1.9; even for the same type of operations, the speed-up variance is also quite high, due to different input sizes. The large variation across operations implies that uniform proportional model replication among devices may not be efficient for training expedition; fine-grained replica allocation at individual operation level could bring more efficient computation power usage for most expedited end-to-end model training.

Tradeoff in communication and computation overhead between DP and MP. Though using model parallelism for some operations eliminates communication of their gradients, there exists data transfer for sending input into operations and dispatching output to other operations. Besides, completion time of the operations is longer, as compared to their parallel execution over multiple devices. It is difficult to decide whether to use DP or MP for an operation, which depends on the amount of data or gradient for transfer, computation power of devices to place the operations, etc.

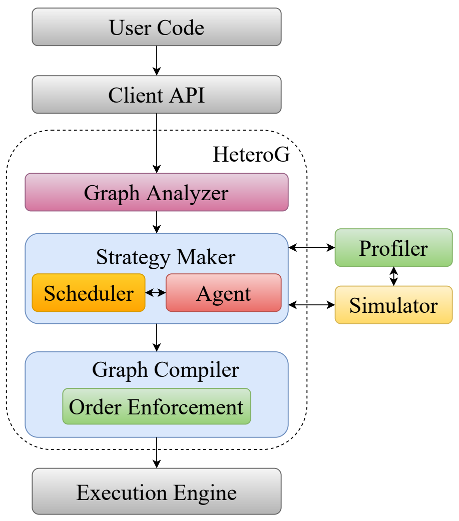


Figure 4: Overall architecture of *HeteroG*.

Tackling these challenges, we carefully design a strategy framework to produce operation-level parallelism, placement, communication and scheduling strategies.

3 SYSTEM DESIGN

3.1 *HeteroG* Overview

HeteroG is designed as a middleware between the client API and core processing engine in a state-of-the-art training framework (e.g., TensorFlow [1], MXNet [5]), to produce the best distributed training scheme for a given DNN model over a set of heterogeneous devices. *HeteroG* takes as input the DAG of the DNN and the device set, and produces a distributed execution graph with operations' device placements, gradient aggregation methods and execution order.

Fig. 4 shows the overall architecture of *HeteroG*. The *Graph Analyzer* analyzes the DNN's computation graph. The *Strategy Maker* runs our strategy framework to generate optimized strategies for operation placement, tensor communication and execution schedule. Then, the *Graph Compiler* applies the strategies to produce the distributed training DAG and enforces execution orders with the execution engine.

To facilitate strategy making with *Agent* and *Scheduler*, the profiler runs different models in the given environment to profile execution time of each operation and transfer time of tensors across different devices; the *Simulator* exploits profiled information to estimate per-iteration training time under different strategies, for Agent's policy learning.

3.2 Graph Analyzer

Graph Analyzer analyzes the original computation DAG, i.e., obtains the `graphdef` of the DNN model, which is a low-level representation of the computation graph regardless of which API is used to build the DAG (e.g., Estimator, Keras, etc.), in case the TensorFlow framework is used.

3.3 Strategy Maker

Our complete set of distributed training strategies includes the following.

(i) Parallelism (DP or MP) and placement for each operation: for DP, an operation is replicated into multiple replicas which are deployed onto multiple devices, with input data evenly divided among the replicas; with MP, the operation is not replicated and deployed onto one single device.

(ii) Gradient communication methods (PS or AllReduce) for gradient aggregation operations.

(iii) Execution order of operations based on placements.

The goal is to minimize per-iteration training time of the DNN, i.e., end-to-end execution time of the respective DAG. The complete problem is very hard in nature: even the subproblem of deciding the execution order of operations within a restricted solution space (i.e., not considering operation replication and communication methods) is already NP-hard, which can be reduced to the DAG task scheduling problem [32] or the job-shop problem [3]. Given the significant hardness of solving the complete problem using a combinatorial optimization approach, we design a novel strategy framework joining both combinatorial optimization and graph neural network (GNN)-based learning to tackle the large strategy space.

We divide the strategy set into two parts and tackle each using a different methodology based on output from each other:

Part-I includes decisions (i) and (ii) which modify the single-GPU computation DAG into a distributed graph;

Part-II includes decisions (iii) for setting the execution order of operations in the distributed training graph.

We adopt a GNN to produce Part-I strategies; we design an efficient heuristic to solve the remaining Part-II problem (which is still NP hard though with a smaller decision space), given Part-I decisions; we compute DAG execution time based on all the decisions and use it as the reward for GNN policy learning. The rationale behind is to pursue an optimization problem (for Part-II decisions) that is close to a known one, with efficient approximation algorithms in place, while using the GNN to produce decisions for harder components of the complete problem.

The Strategy Maker consists of the following components for strategy making:

Agent. The agent runs the GNN, using input feature vector created base on profiling data, and generates Part-I decisions. Details of the GNN design will be introduced in Sec. 4.1.

Scheduler. The scheduler runs the heuristic (Sec. 4.2) to compute execution order of all operations, based on decisions made by the Agent.

Two auxiliary modules are used for building the Agent:

Profiler. It profiles the given DNN model to obtain execution time of each operation on different devices under different batch sizes, the size of the tensor transferred between operations, and the link bandwidth between each pair of devices. We run the given DNN model on each device with different representative batch sizes, if the model can be fit into the device memory. For a large model that cannot be fit into a GPU, we use model parallelism, and try different placements on multiple devices. These allow us to measure computation time of each operation on different devices with different input sizes, so that we can build a linear regression model to predict computation time of a specific operation at other batch sizes, according to the type of operation, the shape of its input, the device that runs the operation, and other attributes of the operation such as the dilation of a `Conv2D` node. For models that can be fit into a single device's memory, it takes less than 10 minutes to complete the profiling; for larger models that cannot be fit into a GPU, the profiling typically takes less than half an hour. We transfer data with different sizes between each pair of devices, record the transfer time and build a linear regression model for transfer time prediction over each link based on the size of tensor for transfer.

Simulator. The simulator is used for training the GNN in the *Agent*. It simulates training according to the strategies produced by the *Agent* and the *Scheduler*, using profiled data from the *Profiler*. It estimates the per-iteration training time for setting rewards for GNN training, and also tracks memory usage on each device, to set bad rewards for strategies leading to memory overflow.

3.4 Graph Compiler

The *Graph Compiler* receives strategies produced by the *Strategy Maker* and generates a distributed training model which can be directly run in the heterogeneous environment.

Operation replication. For operations that use DP, Graph Compiler creates replicas of the operation and places them onto the devices (i.e., by setting the 'device' attribute of the node as in TensorFlow). The number of replicas placed on each device is decided by the *Agent*.

Gradient Aggregation. When the PS architecture is chosen for parameter synchronization among replicas of an operation, one device (where a replica of the operation is deployed) performs as the PS as well (to reduce some gradient communication overhead), storing the parameters; gradients from other replicas are sent to the PS. The PS device is chosen as one that minimizes completion time of gradient aggregation.

When AllReduce is selected, gradients are synchronized among all replicas of the operation using an Allreduce algorithm: ring-based AllReduce [36, 45], or a hierarchical AllReduce structure that aggregates gradients among GPUs on the same physical server first and then across servers. We always use the better structure among the two by estimating the communication time of the two based on the given network topology.

We adopt synchronous SGD for DNN training in *HeteroG*: after gradient aggregation, updated parameters are applied to all replicas. Consequently, parameters are consistent among all model replicas, and the accuracy of the trained DNN model is not affected regardless of the model transformation.

Order Enforcement. Each operation in the distributed training graph is assigned with a priority according to the execution order computed by the Scheduler, for the execution engine to schedule the operations accordingly.

3.5 Client API

HeteroG provides a simple programming interface `get_runner` for developers to call after they build the single-GPU graph. As shown in Fig. 5, `get_runner` accepts as arguments a single-GPU graph (generated by `model_func`), input dataset (`input_func`), device information (`device_info`) including IP addresses (or hostnames)

```
 1 import heterog
 2
 3 def model_func():
 4    #create single GPU model
 5    loss = ...
 6    train_op = ...
 7    return train_op
 8
 9 def input_func():
10    #create input dataset
11    dataset = ...
12    return dataset
13
14 dist_runner = heterog.get_runner(
15    model_func,
16    input_func,
17    device_info,
18    heterog_config)
19
20 dist_runner.run(steps)
```

Figure 5: *HeteroG* programming interface.

of machines and GPU IDs, and an optional *HeteroG* configuration object (`heterog_config`) containing extra arguments if needed (e.g., a file path to save trained variables, whether to use default execution order or our order scheduling algorithm). A developer can first define a single-GPU computation model (lines 3-7) and input dataset (lines 9-12), and then invoke `get_runner` (lines 14-18). The API computes deployment strategies and produces the distributed training model; the returned `dist_runner` object contains the modified graph and its `run` function executes the modified training model according to the execution order, with a maximum number of training steps specified by the developer (line 20).

4 STRATEGY FRAMEWORK

An illustration of our strategy framework is given in Fig. 6.

4.1 GNN-based Policy Learning

We adopt a GNN for Part-I strategy making due to close relation of our decisions with the structure of the DNN graph: embeddings produced by a GNN encode features of the DAG and have been shown effective in facilitating graph-related decision making [2, 64]. We do not use a GNN to produce all strategies as execution order decisions are hard to be described as GNN output (because execution order decisions are for operations on distributed graph rather than original single-GPU graph) and the action space would be too large to learn.

4.1.1 Model Feature Encoding

Different DNN models have different numbers of operations. A GNN is used for creating a flat feature vector for each DNN model, by encoding the graph information into a set of embeddings.

Per-node embeddings. We employ a graph attention neural network (GAT) [53], which achieves better performance than GCN [11, 55, 57] when handling graph-based problems, by aggregating features among neighbors based on correlation coefficient between each pair of feature vectors and using multi-head mechanism to enhance aggregation performance. The GAT takes as input the DAG of DNN model, in the form of: (1) a node feature matrix, where each row contains the operation's attributes (e.g., execution time when running on different devices, the input and output sizes, the average tensor transfer time between each pair of devices);[1] (2) an adjacency matrix describing data dependencies. It generates a per-node embedding vector e_o, by encoding attributes of immediate neighbors of o using multi-head attention layers:

$$\mathbf{e}_o = \|_{k=1}^{K} \sigma\Big(\sum_{j \in \mathcal{N}_o} \alpha_{oj}^{k} W^{k} \mathbf{e}_j'\Big)$$

Here K is the number of heads of multi-head attention layer, $\|$ denotes concatenation of the output of each head, σ is non-linear transformation, N_o is the set of neighbors of o including o itself, α_{oj} is the correlation coefficient between feature vectors of node o and node j, W is the weight vector to be learned, and $\mathbf{e}_j'$ is the output embedding of node j from the previous attention layer.

Per-group embeddings. A DNN model typically contains thousands of operations. Making decisions for each of them results in a very large action space, and hence significant challenge in finding good strategies. We therefore further gather multiple nodes into groups, and learn a set of strategies for nodes in the same group, significantly reducing the action space. We design a nearest-neighbor method to decide the groups: If the number of operations exceeds the maximal group number N, we choose the top-N operations with longest average execution time (these operations contribute more to the per-iteration execution time). We group each of the other operations with one of the N operations with the least number of hops in-between (we want nearby operations to have similar strategies to reduce communication overhead and extra split/concat operations). A per-group embedding $\mathbf{g}_i$ is computed by encoding information from all nodes in this group:

$$\mathbf{g}_n = \sigma\Big(\sum_{o \in \mathcal{G}_n} W\mathbf{e}_o\Big)$$

where $\mathcal{G}_n$ contains all the nodes in group n.

4.1.2 Strategy Network

Embeddings of node groups are concatenated into a feature vector, further fed into a strategy network for making Part-I decisions on operation replication/device placement and gradient aggregation method. We employ a Transformer-XL network [8], which has been shown excellent in handling long embeddings (e.g., for language translation).

We encode Part-I decisions as output of the strategy network. An $N \times (M + 4)$-dimensional action space is designed, where M is the number of GPUs. In the $(M + 4)$-dimensional vector for each group, each of the first M elements represents placing operations in this group to the corresponding device using model parallelism (i.e., no replication on the other devices). The last 4 elements correspond to different data parallelism schemes: the four combinations between two replication decisions (replicating the group onto each of the M devices with one replica per device and proportionally placing

[1]We encode the communication cost between each pair of operations into the input feature vector of the GNN. If the bandwidth changes, the input to the GNN changes and the output strategy changes correspondingly.

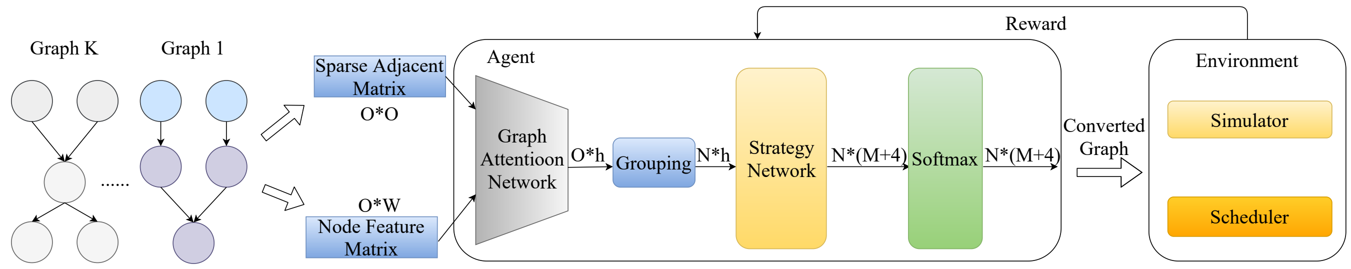


Figure 6: The strategy framework: O is the # of operations in a DNN; W is the dimension of an operation's feature vector; and h is the dimension of the hidden layer output; N is the # of groups; M is the # of GPUs.

a number of replicas of the group onto each device according to computation power) and two communication methods (PS or AllReduce for gradient aggregation in the group). A softmax function is used to produce an action for each group, out of the M+4 strategies.

4.1.3 GNN Training

The graph embedding GAT and strategy network are trained end-to-end together through reinforcement learning (RL) [27]. In each round, a set of DNN graphs, G, are sampled as input to the GAT. For each graph, deployment strategies are produced from the strategy network and a reward is computed by the simulator based on simulated training of the respective DNN using the deployment strategies and execution order (produced by the heurisitc algorithm in Sec. 4.2). The reward is the additive inverse of the square root of the per-iteration execution time of the DNN graph, $R = -\sqrt{T}$, if there is no out of memory (OOM) error; otherwise, we multiply the computed reward by 10, to lower the chance of producing the respective strategy.

The objective of RL is to maximize the overall reward over the $|G|$ input graphs: $J(\theta) = \frac{1}{|G|}\sum_{G}\mathbb{E}_{D\sim\pi_\theta(G)}[R_{G,D}] + \lambda H(\pi_\theta)$, where θ is the set of weights in the GAT and strategy network to learn, and π_θ is the policy distribution to produce actions. The regularization term $H(\pi_\theta)$ [17] allows π_θ to have a high entropy, i.e., high diversity in the decisions, for sufficient exploration of the action space. λ balances exploration and exploitation. With each reward, weights are updated by policy gradients [58]:

$$\theta \leftarrow \theta + \alpha \frac{1}{|G|}\sum_{g}^{|G|} \nabla_\theta \log \pi_\theta(a_g)(r_g - R_g) + \lambda \nabla_\theta H(\pi_\theta)$$

where α is the learning rate, a_g is the action for graph g, r_g is the reward of a_g, and R_g is moving average of the rewards.

4.2 Execution Order Scheduling

Even though the computation operations are already partially ordered based on the data-flow dependency of DAG, there still exist situations that multiple operations placed on the same device are ready to run at the same time, and different orders to execute them may lead to different training time. The scheduler decides the global execution order of all operations (including concat and split) based on the modified training graph after applying the Part-I decisions.

Here, we further treat a link between two GPUs as a device. We regard parameter synchronization among a operation's replicas as a communication operation, and deem that it is placed on a link if the respective PS or AllReduce-based parameter synchronization makes use of the link. Our order scheduling algorithm ensures that every GPU processes at most one computation operation at a time, and every link sends tensor for at most one communication operation at a time.

Our execution order scheduling to minimize per-iteration training time is a combinatorial optimization problem, similar to but simpler than classical task scheduling problems with inter-task dependencies [3] (as the placement of each operation is given). Nonetheless, our problem is still NP-hard, as it is a generalization of the job-shop problem [14]: the job-shop problem schedules tasks with chain-like precedence constraints given their machine placement, while our problem allows arbitrary precedence relations among operations. List scheduling algorithms are commonly used for solving dependency-based task scheduling problems approximately [32]. The core idea of list scheduling algorithms, e.g., HEFT [22], is to assign priorities to tasks, and then assign tasks to the best devices and schedule them on the respective devices in order of their priorities.

We adapt the idea for our execution scheduling. We compute a rank for each operation:

$$\text{rank}(o_i) = p_i + \max_{o_j \in \text{succ}(o_i)} \{\text{rank}\,(o_j)\}$$

where p_i is the computation or communication time of operation o_i, and $succ(o_i)$ is the set of all successors of o_i. Given device placement of the operations, on each device, we order operation execution according to their ranks, and run an operation with a higher rank when it is ready (i.e., its dependencies have all been done), before moving on to the next operation. Multiple devices can execute their respective ready operations concurrently; since we consider inter-GPU links as devices, this maximally allows computation and communication overlap.

We can prove a (tight) performance bound of our order schedule heuristic. Let T_{LS} and T^* be the per-iteration execution time using our heuristic and the ideal optimal schedule, respectively. Recall M is the number of GPUs, and M^2 is the maximal number of links. Detailed proof is in the Appendix.

Theorem 1. *T_{LS} is no larger than $(M + M^2)T^*$.*

Theorem 2. *There exists an instance of our execution order scheduling problem where $\frac{T_{LS}}{T^*} \approx M + M^2$.*

5 IMPLEMENTATION

HeteroG is implemented on Tensorflow 1.14 as a python module that developers can readily import into their Tensorflow code. Core design of *HeteroG* is generally applicable and can be implemented in other ML frameworks as well.

Graph Analyzer is built in Python with 480 LoC.

Strategy Maker. The Agent is implemented in Python with 2156 LoC. We use 12 multi-head attention layers in the GAT, with 8 heads in each layer. The maximum number of groups, N, is 2000. There are 8 layers in the Transformer-XL strategy network.

The Simulator and the Scheduler are built in Rust with 1862 LoC. The simulator simulates training process of the converted DAG. It maintains a ready queue for each device, consisting of operations assigned to the device in computed execution order, whose dependencies have been cleared. It keeps removing an available operation from the head of each ready queue, calculating completion time of the operation according to completion time of its dependencies and the device it is placed on, and adding its child nodes into the ready queue if their dependencies are all cleared. The simulator also simulates memory allocation and releasing when executing an operation (using reference counting), and records the peak memory usage on each of the device. The simulator records the link bandwidth utilization between each pair of devices. When more data are transferred using a specific link, the estimated communication time becomes longer accordingly.

The profiler is implemented based on TensorFlow's built-in profiler by setting the running configuration option `trace_level` to be `FULL_TRACE`. An operation may consist of multiple GPU kernels, the profiler aggregates the execution time of related kernels to obtain an accurate estimation of execution time for each operation. It records the start time of send operator and the end time of the corresponding receive operator, and estimates tensor transmission time as the difference (server clocks are synchronized using NTP).

Graph Compiler generates an executable distributed training model, implemented in Rust with 1051 LoC.

Operation replication. We traverse the nodes, making copies and specifying their 'device' attribute, and then connect them to corresponding copies of inputs. The replica numbers of adjacent nodes (e.g., $o_i \rightarrow o_j$) can be different. For predecessor nodes whose output tensor has the batch size dimension (e.g.,a tensor of dimension $B \times W$ where B is the batch size, we add a `Concat` operation to collect outputs from replicas of o_i and a `Split` node to split it as inputs to replicas of o_j. For other operations whose output does not have the batch size dimension, we do not replicate them.

Gradient Aggregation. For PS-based gradient synchronization, we add a gradient aggregation operation, before an apply gradient operation. For AllReduce, we add collective NCCL primitive operations [23] into the training graph. These NCCL operations receive gradients from operation replicas and handle details of the AllReduce procedure.

An illustration of the original DNN model and converted graph are given in Fig. 7. Operations c, e, h and j adopt DP with extra split and concat operations added; gradient aggregation operations are also added in training graph.

Order enforcement module activates the schedule computed by Scheduler, which is implemented in C++ with 213 LoC. By default, TensorFlow execution engine executes the operations in a ready queue following FIFO (First-In-First-Out). We set each operation

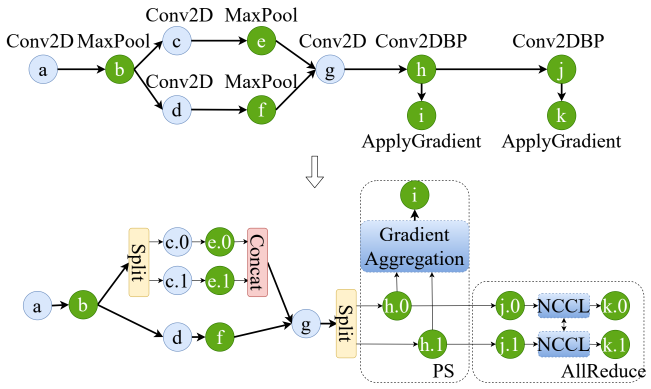


Figure 7: Original and converted DAGs: an example.

with a priority according to the order computed by the Scheduler, such that they will be scheduled by execution engine accordingly.

6 EVALUATION

6.1 Evaluation Methodology

Testbed. We deploy *HeteroG*-boosted TensorFlow framework in 5 physical machines (12 GPUs): one equipped with 4 NVIDIA 16GB Tesla V100 GPUs, two 10-core Intel Xeon processor E5-2630 v4 CPUs and one 100GbE Mellanox RDMA card; two equipped with two 11GB NVIDIA GTX 1080 Ti GPUs, one 8-core Intel Xeon E5-1660 v4 CPU and one 50GbE Mellanox RDMA card; and two equipped with two 12GB NVIDIA Tesla P100 GPUs, one 8-core Intel Xeon E5-1660 v4 CPU and one 50GbE Mellanox RDMA card. The machines are connected through a 100Gbps switch.

Benchmark models. The GNN is trained with 5 types of CNN models (VGG19 [49], ResNet [19], Inception-v3 [50], MobileNet_v2 [46] and NasNet [65]) and 3 types of large NLP models (Transformer [52], Bert-large [9] and Xlnet-large [62]). To evaluate produced strategies, we run real-world distributed training of each DNN model on our testbed according to the strategies.

GNN training process. We profile these benchmark models, generate the adjacency matrix and the feature matrix for each graph as input to the GNN and train the GNN using two Tesla V100 GPUs using data parallelism, which takes around 4 hours to converge. With the trained GNN, *HeteroG* can produce strategies for these models. The GNN model is updated when a new model is provided. We conduct experiments to evaluate the generality of *HeteroG* for unseen graphs in Sec. 6.5, presenting the time taken to update the GNN on a new graph.

Baseline strategies. We compare *HeteroG* with the following baselines. (1) **EV-PS**: data parallelism with one complete model replica per device and PS architecture for gradient synchronization; (2) **EV-AR**: same as EV-PS except for using AllReduce for gradient synchronization; (3) **CP-PS**: data parallelism with the number of model replicas per device proportional to device computation power and using PS for gradient synchronization; (4) **CP-AR**: same as CP-PS except for using AllReduce for gradient synchronization. We also compare performance of *HeteroG* with 4 existing studies on training model deployment: HetPipe [43], FlexFlow [26], Horovod [47] and Post [12].

Table 1: Per-iteration training time (in seconds) of DNN models: *HeteroG* strategies vs. different DP strategies (8 GPUs).

Model (batch size)	*HeteroG*	EV-PS/Speedup	EV-AR/Speedup	CP-PS/Speedup	CP-AR/Speedup
VGG-19 (192)	0.462	0.907 / 96.3 %	0.653 / 41.3 %	0.853 / 84.6 %	0.591 / 27.9%
ResNet200 (192)	0.693	1.431 / 106.4 %	0.955 / 37.8 %	1.273 / 83.7 %	0.897 / 29.4%
Inception_v3 (192)	0.528	0.933 / 76.7 %	0.701 / 32.8%	0.911 / 72.5 %	0.659 / 24.8%
MobileNet_v2 (192)	0.232	0.413 / 78.0%	0.368 / 58.6%	0.394 / 69.8%	0.325 / 40.0 %
NasNet (192)	0.862	1.244 / 44.3 %	1.028 / 19.2%	1.203 / 39.6 %	1.116 /29.5 %
Transformer (6 layers)(720)	0.298	0.961 / 222.4 %	0.496 / 66.4 %	0.931 / 212.4 %	0.361 / 21.1%
Bert-large (24 layers)(48)	0.451	0.612 / 35.7 %	1.064 / 135.9%	0.795 / 76.2%	1.049 / 132.6 %
XlNet-large (24 layers)(48)	0.851	1.232 / 44.8 %	1.551 / 82.2 %	1.283 / 50.8%	1.566 / 84.0 %
ResNet200 (384)	2.285	OOM/-	OOM/-	OOM/-	OOM/-
Transformer (24 layers)(120)	1.147	OOM/-	OOM/-	OOM/-	OOM/-
Bert-large (24 layers)(96)	2.241	OOM/-	OOM/-	OOM/-	OOM/-
XlNet-large (24 layers)(96)	4.254	OOM/-	OOM/-	OOM/-	OOM/-
Bert-large (48 layers)(24)	1.892	OOM/-	OOM/-	OOM/-	OOM/-
XlNet-large (48 layers)(24)	3.468	OOM/-	OOM/-	OOM/-	OOM/-

Table 2: Percentage of operations using different parallelism strategies with *HeteroG* (Gx means placing the operation in the x-th GPU without replication. G0, G1: Tesla V100; G2-G5: GTX 1080Ti; G6, G7: Tesla P100.

Model (batch size)	G0	G1	G2	G3	G4	G5	G6	G7	EV-PS	EV-AR	CP-PS	CP-AR
VGG-19 (192)	2.1%	0	0	0	0	0	0	0	11.7%	28.5%	7.6%	50.1%
ResNet200 (192)	4.2%	0	0	0	0	0	0	0	25.7%	29.2%	8.1%	32.8%
Inception_v3 (192)	1.8%	0	0	0	0	0	0	0	18.4%	21.5%	21.7%	36.6%
MobileNet_v2 (192)	3.7%	0	0	0	0	0	0	0	31.7%	41.2%	9.5%	13.9%
NasNet (192)	3.6%	0	0	0	0	0	0	0	8.8%	66.5%	10.4%	10.7%
Transformer (6 layers)(720)	3.8%	0	0	0	0	0	0	0	12.2%	21.1%	18.2%	44.7%
Bert-large (24 layers)(48)	5.3%	0	0	0	0	0	0	0	41.4%	23.9%	11.1%	18.3%
Xlnet-large (24 layers)(48)	6.8%	5.4%	0	0	0	0	0	0	33.5%	28.1%	11.6%	14.6%

We adopt strong scaling in all experiments with fixed global batch size in case of DP. All results are averaged over 500 iterations.

6.2 Per-iteration Training Speed-up

We first compare the per-iteration time of different models when trained on 8 GPUs (2 Tesla V100, 4 GTX 1080Ti, 2 Tesla P100), using strategies produced by *HeteroG* and four DP baselines. In Table 1, the speed-up is computed by dividing the difference between the two strategies' time by that of *HeteroG*. *HeteroG* outperforms all DP baselines with speed-ups ranging from 19.2% to 222.4%. *HeteroG* achieves 222.4% and 212.4% speed-up over Transformer (6 layers) EV-PS and CP-PS, as communication is heavy when training Transformer and using PS only is less efficient. For NasNet EV-AR where *HeteroG* achieves only 19.2% speed-up, we see in Table 2 that EV-AR is the parallelism strategy of 66.5% operations in NasNet, as selected by *HeteroG*, implying that EV-AR is already a good strategy for NasNet. Besides, when the batch size or a model size become larger, DP becomes infeasible for training large models (out-of-memory), while *HeteroG* can still find feasible solutions.

We record the percentage of operations in each DNN adopting different parallelism strategies, as decided by *HeteroG*, in Table 2. EV-PS, EV-AR, CP-PS and CP-AR represent the DP strategies used by individual operations. We have the following observations.

Hybrid of PS and AllReduce for parameter synchronization. We see that a mixture of PS and AllReduce methods are used for aggregating gradients in each DNN model. Due to NCCL's limitation, AllReduce for different operations cannot be launched simultaneously; with hybrid, PS-based gradient aggregation of some operations can start when a parameter synchronization process using AllReduce is in waiting stage (for receiving gradients from other devices). The communication channel can be better utilized, while GPUs are running computation operations, leading to a better overlap between communication and computation.

Different device distribution of replicas. Table 2 also shows that among operations which use DP, the percentages of having the same number of replicas per device or a proportional number of replicas according to device computation power, are quite comparable. As shown in Sec. 2.3, effectiveness of replicating according to device computation power is different for different types of operations and for the same operation with different input sizes. Consistently here, a hybrid of even replication and proportional replication are chosen for operations within each DNN model.

Eliminating large gradient aggregation. For each DNN model, a small percentage of operations are placed in GPU0 (or GPU1) without replication (i.e., using MP instead of DP). A close inspection reveals that those are mostly operations with a large number of parameters (e.g., operations in the last fully connected layer in VGG-19 and ResNet200, word embedding layer in Bert-large and Xlnet-large and the operations to compute their gradients). When those operations are placed in a single GPU, their gradients do not need to be aggregated and applied to multiple GPUs, so the related communication overhead is eliminated.

Deployment of Large Models. Table 1 has shown that *HeteroG* can find customized strategies to successfully run large models (ResNet200, Bert-large and Xlnet-large with larger batch sizes, Transformer,

Table 3: Percentage of operations using different parallelism strategies with *HeteroG*: large models (Gx means placing the operation in the x-th GPU without replication).

Model (batch size)	G0	G1	G2	G3	G4	G5	G6	G7	EV-PS	EV-AR	CP-PS	CP-AR
ResNet200 (384)	28.7%	21.5%	14.7%	12.8%	6.5%	7.8%	0	0	2.8%	5.2%	0	0
Transformer (48 layers)(120)	33.6%	29.8%	7.2%	6.1%	4.4%	5.6%	2.1%	3.5%	1.7%	1.4%	2.2%	2.4%
Bert-large (24 layers)(96)	20.6%	17.7%	11.5%	10.8%	12.3%	8.2%	5.4%	3.6%	3.3%	2.8%	1.2%	2.6%
Xlnet-large (24 layers)(96)	22.3%	19.4%	9.8%	8.4%	10.3%	8.1%	5.2%	5.8%	2.3%	3.5%	3.3%	1.6%
Bert-large (48 layers)(24)	22.8%	21.6%	8.4%	7.2%	8.7%	7.9%	6.4%	4.3%	2.8%	3.5%	2.7%	3.7%
Xlnet-large (48 layers)(24)	28.6%	24.7%	5.4%	7.5%	6.3%	5.9%	3.3%	3.7%	2.9%	4.1%	2.7%	4.9%

Table 4: Per-iteration training time (in seconds) of DNN models: *HeteroG* strategies vs. different DP strategies (12 GPUs).

Model (batch size)	*HeteroG*	EV-PS/Speedup	EV-AR/Speedup	CP-PS/Speedup	CP-AR/Speedup
VGG-19 (288)	0.503	0.911 / 81.1%	0.682 / 35.6%	0.896 / 78.1%	0.633 / 25.8%
ResNet200 (288)	0.745	1.522 / 104.3%	1.085 / 45.6%	1.298 / 74.2%	0.966 / 29.7%
Inception_v3 (288)	0.641	0.987 / 53.9%	0.806 / 25.8%	0.954 / 48.8%	0.791 / 23.4%
MobileNet_v2 (288)	0.255	0.421 / 65.1%	0.411 / 61.2%	0.403 / 58.1%	0.337 / 32.1%
NasNet (288)	0.915	1.385 / 51.3%	1.123 / 22.7%	1.275 / 39.3%	1.348 /47.3%
Transformer (6 layers)(1080)	0.419	1.133 / 170.4%	0.605 / 44.3%	1.112 / 165.3%	0.547 / 30.5%
Bert-large (24 layers)(72)	0.538	0.825 / 53.3%	1.234 / 129.3%	0.821 / 52.6%	1.218 / 126.4%
XlNet-large (24 layers)(72)	0.972	1.447 / 48.8%	1.681 / 72.9%	1.485 /52.8%	1.832 / 88.5%
ResNet200 (576)	3.031	OOM/-	OOM/-	OOM/-	OOM/-
Transformer (24 layers)(180)	1.544	OOM/-	OOM/-	OOM/-	OOM/-
Bert-large (24 layers)(144)	2.611	OOM/-	OOM/-	OOM/-	OOM/-
XlNet-large (24 layers)(144)	5.043	OOM/-	OOM/-	OOM/-	OOM/-
Bert-large (48 layers)(36)	2.367	OOM/-	OOM/-	OOM/-	OOM/-
XlNet-large (48 layers)(36)	3.812	OOM/-	OOM/-	OOM/-	OOM/-

Table 5: End-to-end training time (in minutes) of DNN models: *HeteroG* strategies vs. different DP strategies.

Models	8GPUs (batch size=192)			12GPUs (batch size=288)		
	HeteroG	CP-PS/Speedup	CP-AR/Speedup	*HeteroG*	CP-PS/Speedup	CP-AR/Speedup
VGG-19	513.1	930.2/81.3%	660.9/28.8%	369.8	667.1/80.4%	457.1/23.6%
ResNet200	633.1	1137.1/79.6%	807.8/27.6%	423.8	726.7/71.8%	533.1/25.8%
Inception_v3	834.6	1463.9/75.4%	1047.5/25.5%	643.6	980.8/52.4%	783.9/21.8%
MobileNet_v2	221.4	369.5/66.9%	319.5/44.3%	169.8	264.5/55.8%	229.7/35.3%
NasNet	1191.3	1683.3/41.3%	1537.9/29.1%	863.9	1179.2/36.5%	1134.3/31.3%

Bert-large and Xlnet-large with more layers), of which pure DP training incurs OOM errors. We further inspect operation-level parallelism decisions made by *HeteroG* for these models in Table 3. Different from results in Table 2 where most operations of the smaller DNN models use DP, most operations in large models in Table 3 are deployed in a single device without replication. This can be explained by the large memory demand of operations in large models, because both more layers and increased batch size lead to increased memory usage.

6.3 Per-iteration Training Speed-up with More GPUs

We next evaluate the per-iteration time when training the DNNs with all 12 GPUs using *HeteroG*. We conduct the same experiments as described in Sec. 6.2. Table 4 shows that *HeteroG* also performs well when scaling to more GPUs. We observe that the speed-up differs compared to using 8 GPUs in Table 1 (e.g., higher speed-up is achieved with NasNet, Bert-large, XLnet, and lower speed-up results with VGG-19, ResNet200, Transformer). With 12 GPUs, the communication time takes a larger portion in the per-iteration training time with DP baselines, especially for models like Bert-large, in which the percentage of computation intensive operators (e.g. Conv2D) is small. *HeteroG* finds optimized strategies to alleviate increased communication in the training time, resulting in higher speed-up as compared to DP baselines. Note that the per-iteration training time in Table 4 is larger than in Table 1, as the global batch size is increased in this set of experiments with more GPUs.

6.4 End-to-End Performance

The end-to-end model training time of the DNN models, for the training to converge to the respective target Top-5 accuracy as reported in the state-of-art benchmarks [7, 49, 51], is given in Table 5. *HeteroG* achieves most expedited training completion as compared to baselines and the speedup is similar to the speedup of per-iteration training time given in Table 1 and 4. This is because in *HeteroG*, our modification of a DNN graph does not change the training semantics of the DNN model, as we only change the number of replicas of operations, their device placement and execution order. The modified DNN training graph is mathematically equivalent to the original DNN model, i.e., the same input leads to the same output [25]. Further, we always adopt the same global batch

Table 6: The time (in minutes) for training GNN to find the best strategy for unseen graph.

Models	From scratch (8GPU/12GPU)	On pre-trained model (8GPU/12GPU)	Ratio (8GPU/12GPU)
VGG-19	82.5/113.4	21.2/25.3	25.7%/22.4%
ResNet200	174.7/201.3	27.3/30.7	15.6% /15.3%
Inception_v3	112.6/141.5	25.1/29.4	22.9%/20.1%
MobileNet_v2	105.2/144.6	26.5/29.8	25.2%/20.6%
NasNet	154.9/191.4	33.4/40.7	21.6%/21.3%
Transformer	143.2/178.8	36.9/41.4	25.8%/23.2%
Bert-large	196.1/243.9	45.1/48.7	22.9%/19.9%
Xlnet-large	211.7/245.3	41.4/46.5	19.5%/18.9%

Table 7: Per-iteration training time (in seconds) with/without *HeteroG* order scheduling (8 GPUs).

Models (batch size)	*HeteroG* Schedule	FIFO Schedule	Speed-up
VGG-19 (192)	0.462	0.512	10.8%
ResNet200 (192)	0.693	0.761	9.8%
Inception_v3 (192)	0.528	0.602	14.1%
MobileNet_v2 (192)	0.232	0.269	15.9%
NasNet (192)	0.862	0.989	14.8%
Transformer (6 layers)(720)	0.298	0.322	11.4%
Bert-large (24 layers)(48)	0.451	0.514	13.9%
Xlnet-large (24 layers)(48)	0.851	1.005	18.1%

size to train a DNN using our modified training graph as when training the DNN using the baselines. As a result, the total number of training iterations needed for model convergence is not changed [48], as compared to using the baselines.

6.5 Generalization to Unseen Graphs

We next evaluate generality of the GNN model in the Agent of Strategy Maker in *HeteroG*. In this set of experiments, we train the Agent using the 8 DNN graphs excluding one graph (which is the unseen graph). In standard GNN-based predictions, only one node in a graph needs to be classified to different classes [10, 63]; given the small action space, prediction based on unseen graphs can be directly done using pre-trained model without further fine-tuning. In our system, the graph structure of different DNNs varies significantly and every node needs to be classified, resulting in a very large action space. In such cases, further GNN fine-tuning is necessary on unseen graphs, as also reported in [2, 20, 64]. Especially, we consider a graph as new/unseen, if the graph's structure is different from graphs that have been used for training.

We experiment with training the GNN from scratch using a single graph, and record the time taken for GNN policy to converge. We also record the time needed for continuing training the pre-trained model on the unseen graph until it converges to the same best strategy found by training from scratch. Table 6 shows that training an unseen graph based on the pre-trained GNN model takes much shorter time than training from the scratch. It indicates that the pre-trained GNN model has indeed learned useful structures from other graphs, and can generalize pretty well to unseen graphs with a small amount of fine-tuning, i.e., the GNN does not need to be re-trained from scratch for an unseen graph. Further, with more GPUs, the fine-tuning time only increases slightly based on the pre-trained model.

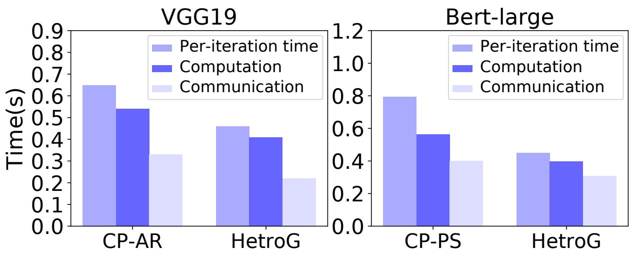


Figure 8: Computation time and communication time per-iteration (8-GPU training).

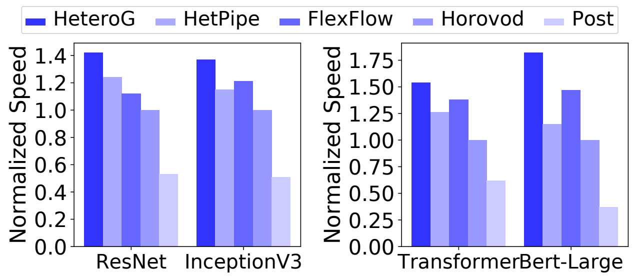


Figure 9: Comparison with existing schemes (12 GPUs).

6.6 Effects of Order Scheduling

We evaluate the effectiveness of our execution order scheduling heuristic (Sec. 4.2), by running model training with our scheduling heuristic in place and with TensorFlow's default FIFO execution schedule. Table 7 shows that our scheduling heuristic accelerates training by about 10~20%.

6.7 Time Breakdown

Fig. 8 shows the average per-iteration training time, computation time and communication time using DP baselines and *HeteroG*. Due to overlap of computation and communication, overall per-iteration training time is usually smaller than the sum of computation and communication time. With *HeteroG*, computation time is smaller than pure DP, due to *HeteroG*'s careful selection of per-operation parallelism and placement strategies; communication time is also reduced because PS or AllReduce is carefully selected and some operations use MP without replication, eliminating gradient aggregation overhead. When training VGG19, the ratio of the sum of computation and communication time over per-iteration training time is 1.31 with CP-AR and 1.47 with *HeteroG*. For Bert-large, the ratio is 1.21 with CP-PS and 1.56 with *HeteroG*. They show that *HeteroG* achieves better overlap of computation and communication.

6.8 Comparison with Existing Studies

Existing works HetPipe [43], FlexFlow [26], Horovod [47] and Post [12] are most relevant to ours. We implemented FlexFlow and Horovod using their open-sourced code and did our best to re-implement Hetpipe and Post according to the algorithms proposed in their papers. We compare the training speed of *HeteroG* with these schemes, when each DNN model is trained on 12 GPUs. Fig. 9 shows the normalized training speeds, computed by dividing the training speed (samples/second) of each scheme by that of Horovod. We observe that the training speed with *HeteroG* is the highest, outperforming other schemes by 16.4% ~ 391.8%. Post only considers operation-to-device placement but not operation-level data parallelism. HetPipe uses heuristics to divide GPUs into multiple virtual workers, utilizes layer-level pipeline parallelism within each virtual worker and data parallelism across different virtual workers, but does not consider operation-level optimization, limiting the solution space. None of the four existing schemes investigate different gradient aggregation methods and execution order of operations. *HeteroG* systematically addresses a large strategy space and achieves better performance.

7 RELATED WORK

Hybrid Gradient Aggregation Methods in DP. Parallax [28] advocates different gradient aggregation methods for different types of parameters in models with large word embedding: PS for sparse parameters and AllReduce for dense parameters. BlueConnect [6], Blink [54] and Plink [33] optimize AllReduce architectures in heterogeneous network environments. They do not address computation power heterogeneity among computing devices.

Device Placement of Deep Learning Models. The Google team uses RL to generate device placement policies for groups of operations in DNN models, with manual operation-to-group assignments [39]; they later propose HDP [38] that jointly learns two NNs for assigning operations to groups and placing groups to devices, respectively. POST [12] integrates an online RL algorithm and a batch learning algorithm, to learn a policy for device placement of DNN operations. Follow-up work have exploited GNNs to learn more general policy networks applicable to different computation graphs [2, 41, 42, 64]. For example, GDP [64] trains a GNN to produce operation-to-device placement for each operation; the action space is much smaller than ours, without deciding operation replication or gradient aggregation methods.

Hybrid Parallelism. Stanza [59], DLPlacer [40] and Alex [30] adopt DP for training convolutional layers in CNN models and MP for other layers. OptCNN [24] parallelizes CNN model training by splitting operations along batch and channel dimensions; training over homogeneous devices is considered. Tofu [56] utilizes a partition-n-reduce method to split a single operation into sub-operations, and a dynamic programming approach to recursively optimize the partition; no device placement of operations is considered. FlexFlow [26] explores the SOAP (Sample-Operation-Attribute-Parameter) search space addressing parallelism within and across operations, and does not consider gradient aggregation methods or execution order of operations.

Pipeline Parallelism. Pipelining has been studied to accelerate DNN training: different DNN layers are deployed on different devices; a mini-batch is divided into micro-batches and the micro-batches can be processed at different devices concurrently [18, 60]. GPipe [21] uses pipelining to address memory bottlenecks for training large NNs. PipeDream [18] introduces a pipelining approach to overlap communication and computation for asynchronous training. HetPipe [43] integrates pipeline parallelism with DP in heterogeneous environments, making layer-based parallelism decisions but not operation-level without considering execution ordering.

With pipeline parallelism, semantics of the original model training is often not retained: pipelining results in multiple versions of parameters during training (similar to asynchronous training), which may lead to more training steps for the model to converge to an acceptable accuracy, or convergence to a different accuracy. *HeteroG* accelerates training while retaining exactly the same semantics as single-GPU model training, through ensuring synchronized parameter updates at all operations. If retaining model training semantics was not a concern, *HeteroG* can be readily integrated with a pipelining design: after producing the distributed training graph, we can further split a mini-batch into micro-batches, carry out pipelined training across operations deployed on different devices, and augment our execution order scheduling algorithm to handle such micro-batches.

Deep Learning in Heterogeneous Environment. Kim *et al.* [29] propose a hierarchical aggregation method based on data parallelism in a heterogeneous GPU cluster: AllReduce architecture for GPUs within the same server and PS architecture among different servers. It does not consider fine-grained operation-level hybrid data and model parallelism. Prague [34] proposes a novel communication primitive, Partial AllReduce, to accelerate asynchronous training in heterogeneous environments. *HeteroG* focuses on synchronous training with ensured model accuracy.

Multi-job Scheduling in Deep Learning Clusters. Gandiva [61] is a cluster scheduling framework that utilizes domain-specific knowledge to improve latency and efficiency of training models in a GPU cluster; it exploits intra-job predictability to time-slice GPUs efficiently across multiple jobs. $Gandiva_{fair}$ [4] and Themis [35] propose schedulers that balance conflicting goals of efficiency and fairness in GPU clusters. Tiresias [16] schedules DL jobs to reduce their job completion time. Some works study algorithms for multi-dimensional resource packing for multi-job scheduling [15, 35, 37]. These studies focus on multi-job scheduling/placement in a cluster, which is orthogonal to *HeteroG*, as *HeteroG* focuses on single training job acceleration. For multi-job scheduling, *HeteroG* can be used as a blackbox, feeding in resource provisioning to a job and obtaining the training speed of the job based on produced strategies; then we can balance resource allocation to different jobs, to achieve targeted global objectives such as fairness, maximal resource utilization or job completion time minimization.

8 CONCLUSION

We present *HeteroG*, an automated module to incorporate with existing machine learning frameworks for DNN training acceleration in heterogeneous GPU clusters. *HeteroG* advocates operation-level hybrid parallelism, communication architecture selection and execution scheduling, based on a carefully designed strategy framework exploiting both GNN policy learning and combinatorial optimization. It achieves up-to 222% training speed-up as compared with various existing data-parallel and hybrid parallel training schemes. *HeteroG* also enables efficient training of large models over a set of heterogeneous devices where simple data parallelism is infeasible. We show that a hybrid of DP and MP, variable device distribution of replicas, mix of PS and AllReduce for parameter synchronization and a close-to-optimal execution schedule together are critical for excellent distributed training performance in heterogeneous clusters.

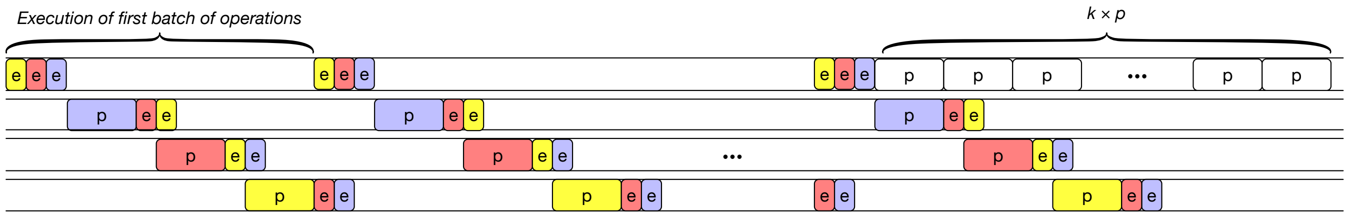


Figure 1: Worst-case instance scheduled by *LS*.

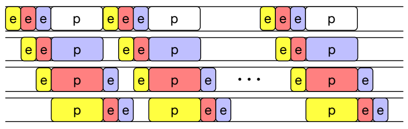


Figure 2: Optimal schedule for the worst-case instance.

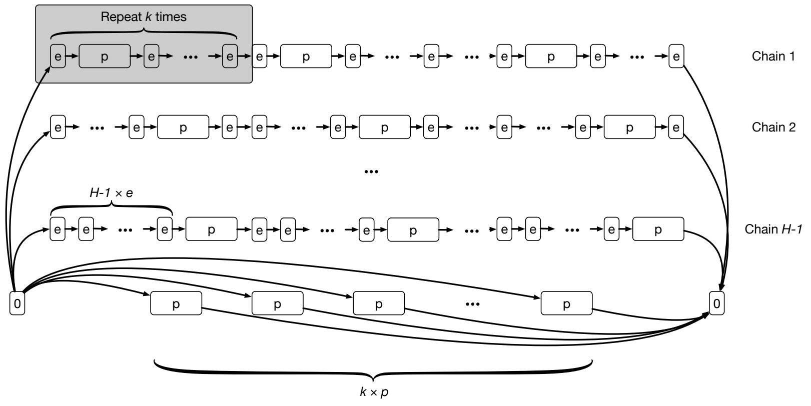


Figure 3: A worst-case instance.

APPENDIX

Upper Bound of Order Scheduling Algorithm

We define the total per-iteration time using the algorithm as T_{LS} and the optimal per-iteration time as T^*.

Theorem 1. *T_{LS} is no larger than $(M + M^2)T^*$*

Proof. We use O to denote the set of all computation and communication operations. We have:

$$T_{LS} \le \sum_{o_i \in O} p_i \le (M + M^2)T^*$$

The first inequality is due to that T_{LS} is no larger than the total amount of the data transmission and execution time of all operations. The second inequality is because that T^* is greater than the total operation execution/transmission time on any device, making $T^* \ge \frac{\sum_{o_i \in O} p_i}{M+M^2}$.

□

Furthermore, we craft a worst-case DAG instance in the following Theorem 2.

Theorem 2. *There is an instance for which $\frac{T_{LS}}{T^*} \approx M + M^2$.*

Proof. For ease of representation, we let H equal to $M + M^2$. Let us consider the following instance in Fig. 3, where p and e are the corresponding operation execution or transmission time with e close to 0. Ignoring the dummy root and sink operation, the DAG is mainly composed of $H - 1$ chains and k individual p operation. Each chain contains kH operations with $(n * H + h)^{\text{th}}$ operations placed on device h, $h \in [H]$. The k individual p operations are all placed on device H. We assume $k \gg H$.

We assume the execution time for the root and sink operation to be 0. Since the rank for the first operation of each chain is larger than the k operations, the chains are to be executed first.

Let us first consider the first H operations of each chain, denoted as the first batch of operations. We denote the i^{th} operation of chain j as $o_{i,j}$. Since all rank$(o_{1,j})$, $j \in [H - 1]$ are the same, we let the execution order on device 1 to be from $o_{1,H-1}$ to $o_{1,1}$. On the second device, the rank for all operations are also the same, and we let the execution order on device 2 to be from $o_{2,1}$ to $o_{2,H-1}$. For operations with the same rank on all other devices, we ensure the execution order is from chain 1 to the last chain. Consequently, we shall execute all $(H - 1) \times H$ operations but the last operations in chain 1 to $H - 2$ in $(H - 1)p + (2H - 3)e$. The execution time for the last operations in chain 1 to $H - 2$ can be overlapped by the execution of next batch of operations, i.e., the second H operations of each chain.

Similarly, the second batch of operations to the second last batch of operations require $(H - 1)p + (2H - 3)e$ to execute respectively. The last batch of operations along with the k p individual operations are executed using $(H - 1)e + kp$. Therefore, the total per-iteration time for this case is:

$$\begin{aligned} T_{LS} &= (k-1)((H-1)p + (2H-3)e) + (H-1)e + kp \\ &= ((k-1)H + 1)p + ((k-1)(2H-3) + H - 1)e \end{aligned}$$

However, the optimal execution time is:

$$T^* = k(p + (H-1)e) + (H-2)e$$

Ensuring $e \longrightarrow 0$ and k to be large enough, we have:

$$\frac{T_{LS}}{T^*} = \frac{((k-1)H+1)p + ((k-1)(2H-3) + H - 1)e}{k(p + (H-1)e) + (H-2)e} \approx H = M + M^2$$

We illustrate the case of scheduling the DAG instance on four devices by LS in Fig 1, and the optimal schedule in Fig 2, where the three chains are colored with purple, red and yellow. We ignore the dummy root and sink operations in the illustration.

□

REFERENCES

[1] 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 *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[2] 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*.

[3] David Applegate and William Cook. 1991. A computational study of the job-shop scheduling problem. *ORSA Journal on computing* (1991).

[4] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*.

[5] 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).

[6] Minsik Cho, Ulrich Finkler, Mauricio Serrano, David Kung, and Hillery Hunter. 2019. BlueConnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *IBM Journal of Research and Development* (2019).

[7] Cheng Cui, Zhi Ye, Yangxi Li, Xinjian Li, Min Yang, Kai Wei, Bing Dai, Yanmei Zhao, Zhongji Liu, and Rong Pang. 2020. Semi-Supervised Recognition under a Noisy and Fine-grained Dataset. *arXiv preprint arXiv:2006.10702* (2020).

[8] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860* (2019).

[9] 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).

[10] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph neural networks for social recommendation. In *The World Wide Web Conference*.

[11] Yixiong Feng, Kangjie Li, Yicong Gao, and Jian Qiu. 2020. Hierarchical graph attention networks for semi-supervised node classification. *Applied Intelligence* (2020).

[12] 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.

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

[14] Leslie Ann Goldberg, Mike Paterson, Aravind Srinivasan, and Elizabeth Sweedyk. 2001. Better approximation guarantees for job-shop scheduling. *SIAM Journal on Discrete Mathematics* (2001).

[15] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review* (2014).

[16] 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)*.

[17] Yong Guo, Yin Zheng, Mingkui Tan, Qi Chen, Jian Chen, Peilin Zhao, and Junzhou Huang. 2019. Nat: Neural architecture transformer for accurate and compact architectures. In *Advances in Neural Information Processing Systems*.

[18] 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).

[19] 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.

[20] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. 2019. Strategies for Pre-training Graph Neural Networks. *arXiv preprint arXiv:1905.12265* (2019).

[21] 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).

[22] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D Anger, and Chung-Yee Lee. 1989. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.* (1989).

[23] Sylvain Jeaugey. 2017. Nccl 2.0. *GTC* (2017).

[24] 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.

[25] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.

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

[27] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* (1996).

[28] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. 2019. Parallax: Sparsity-aware Data Parallel Training of Deep Neural Networks. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 43.

[29] Youngrang Kim, Hyeonseong Choi, Jaehwan Lee, Jik-Soo Kim, Hyunseung Jei, and Hongchan Roh. 2019. Efficient Large-Scale Deep Learning Framework for Heterogeneous Multi-GPU Cluster. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE.

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

[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] Shi Li. 2020. Scheduling to minimize total weighted completion time via time-indexed linear programming relaxations. *SIAM J. Comput.* (2020).

[33] Liang Luo, Peter West, Arvind Krishnamurthy, Luis Ceze, and Jacob Nelson. 2020. PLink: Discovering and Exploiting Datacenter Network Locality for Efficient Cloud-based Distributed Training. (2020).

[34] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. 2020. Prague: High-Performance Heterogeneity-Aware Asynchronous Decentralized Training. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[35] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient {GPU} Cluster Scheduling. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*.

[36] Amith R Mamidala, Jiuxing Liu, and Dhabaleswar K Panda. 2004. Efficient Barrier and Allreduce on Infiniband clusters using multicast and adaptive algorithms. In *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No. 04EX935)*.

[37] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*.

[38] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. 2018. A hierarchical model for device placement. (2018).

[39] 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.

[40] 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).

[41] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. 2019. Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs. *arXiv preprint arXiv:1905.02494* (2019).

[42] 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).

[43] Jay H Park, Gyeongchan Yun, Chang M Yi, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. 2020. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of

Pipelined Model Parallelism and Data Parallelism. *arXiv preprint arXiv:2005.14038* (2020).

[44] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[45] Pitch Patarasuk and Xin Yuan. 2007. Bandwidth efficient all-reduce operation on tree topologies. In *2007 IEEE International Parallel and Distributed Processing Symposium*.

[46] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*.

[47] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[48] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv* (2019), arXiv–1909.

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

[50] 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*.

[51] Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Hervé Jégou. 2019. Fixing the train-test resolution discrepancy. In *Advances in Neural Information Processing Systems*. 8252–8262.

[52] 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*.

[53] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[54] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. 2019. Blink: Fast and generic collectives for distributed ml. *arXiv preprint arXiv:1910.04940* (2019).

[55] Lei Wang, Yuchun Huang, Yaolin Hou, Shenman Zhang, and Jie Shan. 2019. Graph attention convolution for point cloud semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[56] 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.

[57] Xiang Wang, Xiangnan He, Yixin Cao, Meng Liu, and Tat-Seng Chua. 2019. Kgat: Knowledge graph attention network for recommendation. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

[58] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* (1992).

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

[60] 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).

[61] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*.

[62] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*.

[63] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2018. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434* (2018).

[64] 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).

[65] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*.