Optimizing DNN Compilation for Distributed Training With Joint OP and Tensor Fusion

Xiaodong Yi, Shiwei Zhang, Lansong Diao, Chuan Wu, *Senior Member, IEEE*, Zhen Zheng, Shiqing Fan, Siyu Wang, Jun Yang, and Wei Lin

Abstract—This article proposes *DisCo*, an automatic deep learning compilation module for data-parallel distributed training. Unlike most deep learning compilers that focus on training or inference on a single device, *DisCo* optimizes a DNN model for distributed training over multiple GPU machines. Existing single-device compilation strategies do not work well in distributed training, due mainly to communication inefficiency that they incur. *DisCo* generates optimized, joint computation operator and communication tensor fusion strategies to enable highly efficient distributed training. A GNN-based simulator is built to effectively estimate per-iteration training time achieved by operator/tensor fusion candidates. A backtracking search algorithm is driven by the simulator, navigating efficiently in the large strategy space to identify good operator/tensor fusion strategies that minimize distributed training time. We compare *DisCo* with existing DL fusion schemes and show that it achieves good training speed-up close to the ideal, full computation-communication overlap case.

Index Terms—Distributed systems, machine learning

1 Introduction

Deep learning (DL) compilers have been studied in recent years for deep neural network (DNN) model graph optimization and training (or inference) expedition, e.g., TVM [1], MLIR [2], Relay [3] and XLA [4]. The DL compilers take as input the model definitions in the respective DL framework (e.g., TensorFlow [5], MXNet [6]), and generate code implementation of the models on different types of DL hardware. The transformation from model definition to specific code implementation is highly optimized based on the model specification and hardware architecture, using methods including: (i) front-end optimization such as NOP elimination, zero-dim-tensor elimination, algebraic simplification, operator (op) fusion and layout transformation [7]; and (ii) backend optimization, e.g., loop-oriented optimizations, hardware intrinsic mapping and memory latency hiding [1].

Most of the existing DL compilers focus on accelerating DL model execution on a single device. In distributed training, communication among devices for parameter synchronization plays a key role in dictating the training time and resource (computation device, network bandwidth) efficiency. Compilation optimization for single-device training (e.g., op fusion) may delay inter-device communication, leading to poor computation-communication overlap and hence low distributed training efficiency (Section 2.4).

Currently, only a few projects study compilation optimization in the distributed setting. GShard [8] extends the XLA compiler for distributed training and provides an elegant way to express a wide range of parallel computation patterns. Boehm et al. [9] use enumeration tree search with structural pruning techniques for op fusion, for learning traditional machine learning (ML) models. However, they do not consider op fusion jointly with communication overhead in the distributed environment.

There are also projects focusing on model parallelism and pipeline parallelism. Megatron-LM [10] introduces an efficient intra-layer model-parallel approach to support training of very large transformer models. GPipe [11] and Pipedream [12] propose pipeline parallelism to further improve model parallelism, by pipelining forward computation and backward propagation across several micro-batches. CoCoNet [13] enables optimization of data-, model- and pipeline-parallel workloads in large language models by introducing a domain-specific language that easily expresses distributed training of models.

This paper focuses on front-end compilation optimization to expedite synchronous data-parallel training. Op fusion strategies have been studied as one of the most important optimization methods to reduce computation overhead [4], [14], [15]. Tensor fusion has been shown to play an important role in reducing the communication overhead [16], [17], [18]. We inspect the performance trade-off caused by op fusion and tensor fusion in distributed

- *Xiaodong Yi, Shiwei Zhang, and Chuan Wu are with the Computer Science, University of Hong Kong, Hong Kong. E-mail: {xdyi, swzhang, cwu}@cs.hku.hk.*
- *Lansong Diao, Zhen Zheng, Shiqing Fan, Siyu Wang, and Wei Lin are with the Alibaba Cloud Intelligent, Alibaba Group, Hangzhou, Zhejiang 310052, China. E-mail: {lansong.dls, shiqing.fsq, siyu.wsy, weilin.lw}@alibaba-inc.com, zzchman@gmail.com.*
- *Jun Yang is with the Compute Arch, NVIDIA Corp, Beijing 201210, China. E-mail: yangjunpro@gmail.com.*

Manuscript received 1 December 2021; revised 14 August 2022; accepted 14 August 2022. Date of publication 25 August 2022; date of current version 6 September 2022.
This work was supported in part by Alibaba Group through Alibaba Innovative Research (AIR) Program, and in part by Hong Kong RGC contracts HKU under Grants 17204619 and 17208920.
(Corresponding author: Chuan Wu.)
Recommended for acceptance by J. Zola.
Digital Object Identifier no. 10.1109/TPDS.2022.3201531

training, and advocate joint op and tensor fusion optimization. We propose *DisCo*, an automatic module to jointly optimize computation and communication fusion over a whole distributed DNN training graph. Existing rule-based op fusion strategies rely heavily on expert experience, and are often less than optimal due to limited exploration of the solution space. *DisCo* adopts a search-based algorithm to identify optimized joint fusion strategies. We summarize main contributions of *DisCo* in the following:

▷ We propose an automatic compilation module to jointly optimize op and tensor fusion for distributed training of DNN models, that expedites computation and communication separately while maximally overlapping their execution.

▷ Op fusion and tensor fusion, two conventionally separated optimization passes, are unified into a joint strategy space. A backtracking search algorithm is designed to efficient prune the large strategy space to identify op/tensor fusion solutions that jointly minimize distribution DNN training time.

▷ A *Fused Op Estimator* is built based on a graph neural network (GNN) model to predict the execution time of fused ops. An efficient simulator is created to estimate the end-to-end execution time of a distributed DNN training graph using the Fused Op Estimator, and serves as a cost model to our search algorithm.

▷ We implement *DisCo* based on JAX [19], an XLA-based framework for generating high-performance accelerator code in a manner completely transparent to DNN model developers. To use *DisCo*, a developer only needs to specify two environment variables, not changing a single line of their model code. *DisCo* is open-sourced at https://github.com/TPDS-Submission/Disco

▷ We carry out extensive experiments training state-of-the-art DNN models in GPU clusters, and carefully compare *DisCo* with existing DL fusion schemes. *DisCo* achieves up-to 26.73% training acceleration, close to the maximal speed-up achievable with ideal, full computation-communication overlap. Interestingly, we observe that our joint op and tensor fusion optimization not only increases communication-computation overlap, but also reduces computation time and communication time, separately, as compared to representative single-device fusion designs.

2 Background and Motivation

2.1 Deep Learning Compilation

To alleviate the dependence on customized DL libraries and the burden of manually optimizing DL models on each type of hardware, domain specific DL compilers have been built [1], [4], [20], [21]. DL compilers incorporate DL-oriented optimizations such as layer and op fusion, to generate highly efficient code for training or inference. Similar to traditional compilers, DL compilers utilize *intermediate representation* (IR) as the abstraction of a DNN model for optimization, including high-level IR which represents the control flow and the dependency among the operators and the data, and low-level IR which reflects hardware characteristics such as memory allocation. DL compilers adopt the layered design, including the front-end optimization (transforming the DNN model into the high-level IR and performing graph-level optimization such as dead code elimination and op fusion) and the back-end optimization (transforming the high-level IR into low-level IR and performing hardware-specific optimization). Our study focuses on high-level IR optimization at the DNN graph level.

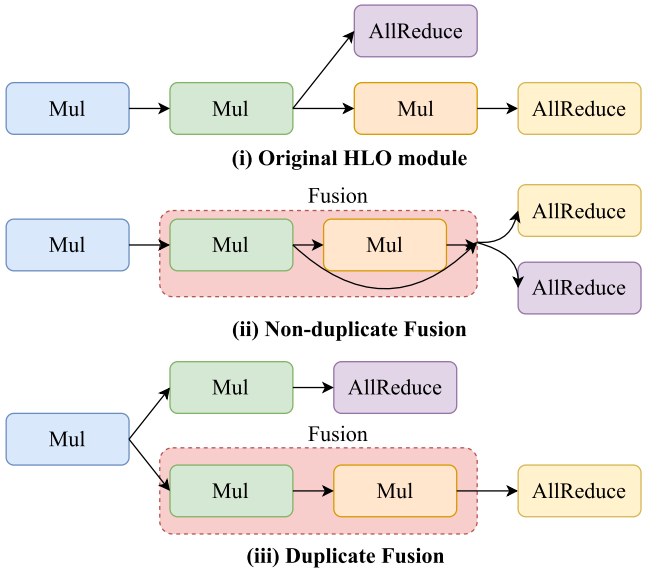


Fig. 1. Non-duplicate fusion and duplicate fusion. An arrow represents gradient/activation passing.

2.2 Computation Operator Fusion

Op fusion [1], [7], [9] is a graph-level optimization that combines multiple computation operators into a single kernel without storing the intermediate results in device memory (e.g., global memory on a GPU). It enables better utilization of computation devices, eliminates device memory allocations for intermediate results, and reduces kernel launch and synchronization overhead, leading to substantially reduced model execution time. Op fusion has been enabled in a number of DL libraries such as TensorFlow XLA [4], Intel Nervana Graph [22] and TVM [1].

To carry out op fusion, typically an op is selected, and then one of its predecessor ops (whose output this op consumes) is chosen to fuse with this successor op. If the chosen predecessor op has multiple successor ops, two main fusion approaches exist: non-duplicate fusion and duplicate fusion [7] (exemplified in Fig. 1). With non-duplicate fusion, the predecessor op is directly fused into the successor op; the output of the predecessor (e.g., gradients) is available for other ops only after the completion of the fused op. With duplicate fusion, the predecessor op is not only fused into the successor op, but also recomputed outside the fused op (so that its output can become available earlier). Op fusion can be carried in a recursive manner over the entire DNN graph: a fused op can be further fused with its predecessor or successor, using a duplicate or non-duplicate fusion approach.

The order of ops to consider for fusion is typically determined by heuristics or learning-based methods [23], [24]. For example, in XLA, ops are chosen according to a predefined post order, and any device memory and computation savings due to fusing the op with a selected predecessor op are evaluated. Such op order-based fusion may not

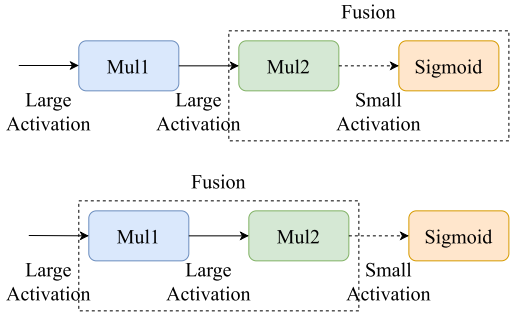


Fig. 2. A case in RNNLM: the order of ops to consider for fusion influences performance significantly.

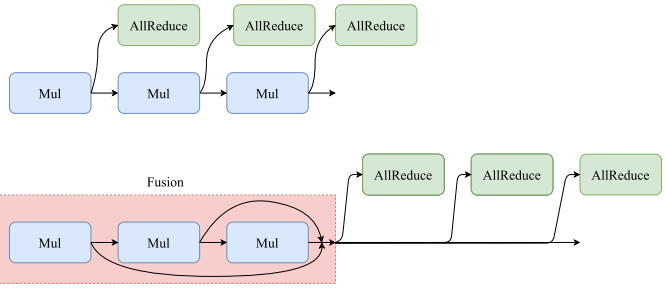


Fig. 3. Delayed communication due to op fusion.

be effective as earlier fusion of some ops may prevent better fusion opportunities for ops considered later. Consider a case of fusing two ops in RNNLM [25] in Fig. 2, where an element-wise multiplication op (Mul1) produces large activations to another multiplication op (Mul2) and Mul2 produces small activations to a Sigmoid function. If Sigmoid ranks higher in the op ordering and is fused with Mul2, the performance does not improve much, since the size of intermediate data (activations) transferred between on-chip memory (local memory for the execution thread) and device memory does not change significantly. If Mul1 and Mul2 are fused instead, activations produced by Mul1 remain in on-chip memory for Mul2, substantially reducing data transfer to/from device memory.

Besides, majority of the existing op fusion systems focus on single-device DNN graph optimization [1], [3], [22], [26].

2.3 Communication Tensor Fusion

In distributed training, data parallelism has been most widely adopted in practice. The training dataset is partitioned into mini-batches at each device. In each training iteration, each worker (device) maintains a replica of the DNN model and carries out Forward Propagation (FP) and Backward Propagation (BP) computation on a mini-batch; gradients from different devices are aggregated before being applied to update model parameters. We focus on accelerating data-parallel training.

AllReduce is a collective instruction, popularly used for parameter synchronization in data-parallel training. It sums (or averages) the gradients from all devices using a ring or tree based algorithm [27], and disperses the aggregated gradients to the devices for parameter update [28]. Commonly one AllReduce instruction is carried out for each gradient tensor produced; the default sizes of tensors in existing DL frameworks (e.g., TensorFlow, PyTorch) may not be ideal for efficient bandwidth utilization. There are usually a large number of small tensors (e.g., over 50% communication tensors in ResNet50 [29] and Transformer [30] are less than 1 MB in size [16]). Such small tensors incur large communication overhead in relation to the short transmission time, e.g., time spent on negotiation/synchronization among workers before actual gradient transfer, which is especially substantial in view of the strict synchronization among workers required by AllReduce.

Tensor fusion advocates fusing multiple small gradient tensors together before executing the AllReduce instruction on the fused tensor. The size of the fused tensor is the sum of sizes of the small tensors. Tensor fusion leads to better bandwidth utilization (with less communication overhead relative to actual gradient transfer); however, start time of the fused AllReduce is delayed, leading to a trade-off effect on the training time.

2.4 Opportunities

To accelerate data-parallel training, existing proposals improve the computation graph on each device with single-device compilation optimizations (e.g., op fusion) and optimize inter-device communication, separately [4], [19]. Op fusion effective on individual devices may be non-optimal or even bring no benefit in distributed training. Op fusion typically merges as many ops as possible to reduce device memory usage and kernel launches; the output of those ops may only be available after the fused op is completely executed. For example, gradients produced by backward propagation ops need to be transferred to other devices for aggregation; fusion of BP ops may delay gradient communication, leading to less computation-communication overlap and hence longer resource idling time.

Fig. 3 gives an example. Suppose the three `Mul` ops are fused, such that AllReduce instructions of gradients produced by the 3 ops are delayed until after the fused op is done. If the delay exceeds computation time reduction, such op fusion increases the training time.

Only a few systems enable DL compilation in distributed training. Based on AllReduce primitives provided by XLA [4], JAX [19] groups the whole processing logic of a DNN model, including the AllReduce instructions, into a single High Level Optimizer (HLO) module (a high-level IR defined in XLA). JAX currently only supports multi-node training across TPU servers rather than GPU servers, and uses rule-based heuristics for op fusion. Rule-based op fusion highly depends on expert experience, and the rule suitable for one model may not fit other models [7]. Further, its computation optimization is separated from communication optimization: op fusion is first conducted, and AllReduce combiner optimization (combining multiple AllReduce instructions together based on a pre-defined tensor size threshold) is performed after op fusion optimization is done. It has been reported [31], [32] that directly applying XLA in distributed training may prolong per-iteration training time (20% slower when training a transformer-based NMT model with Adam Optimizer [31]), as compared to not applying XLA, since communication can be seriously delayed.

There is a trade-off between computation efficiency and communication channel utilization in distributed training,

when both op fusion and tensor fusion are adopted. We advocate a search algorithm to jointly optimize op and tensor fusion, striking a good balance between computation and communication efficiencies and achieving overall training acceleration.

2.5 Challenges

Exploring the opportunities comes with challenges.

Large Search Space for Joint Fusion. A DNN model usually consists of thousands of computation and communication instructions, resulting in a huge search space with various op/tensor fusion combinations. Naive enumeration of possible solutions without pruning is infeasible. We design an efficient backtracking algorithm to prune the search space effectively.

Time- and Resource-Consuming to Evaluate Search Candidates in Real Environments. Unlike single-device training, evaluating each possible solution produced by the search algorithm by running the modified DNN model in a real distributed environment is time- and resource-prohibitive. We build an efficient simulator to estimate the execution time of possible strategies produced by the search algorithm, eliminating the need of heavy real-world trial runs.

Difficulty in Accurate Execution Time Prediction of Fused Ops. Simulators have commonly been used to predict DNN training time under different device placements [26], [33] or with different execution scheduling strategies [34], [35], based on profiled execution time of individual ops. In our case, fused ops that have never been seen before may well be produced. Execution time estimation for fused ops is not easy: even if execution time of each original op is profiled, the interaction among these ops is unknown, and cannot be profiled unless we implement every unseen fused op. Further, execution time of fused op is tightly related to the architecture of the processor, as well as the back-end optimization applied during compilation such as loop fusion, tiling and loop unrolling [7]. Architectural features and compiler code generation interact in extremely complex ways [1], [4], [14]. It is very hard to build a white-box analytical model describing details of the processor or effects of all compiler passes, and their interactions.

We design a GNN-based model for execution time prediction of fused ops. GNNs have been adopted and achieved satisfying performance for various graph-based learning purposes, [36], [37], [38], [39], [40], [41]. It takes as input graph-structured data and learns the structural information based on graph connectivity and node/edge features. We exploit a GNN to learn the execution time from the op fusion graph. We focus on optimizations that preserve model accuracy (exactly the same before and after optimization), and hence do not consider staleness options which may compute gradients based on the last round of weights while communicating the gradients of this round [12].

3 System Design

3.1 *DisCo* Overview

DisCo is designed as an optimizer for TensorFlow XLA's HLO IR, to produce optimized fusion strategies for both computation operators and communication tensors. *DisCo* takes as input the HLO module of a whole DNN model, and produces an optimized HLO module for further back-end compilation optimization. Fig. 4 shows the overall architecture of *DisCo*.

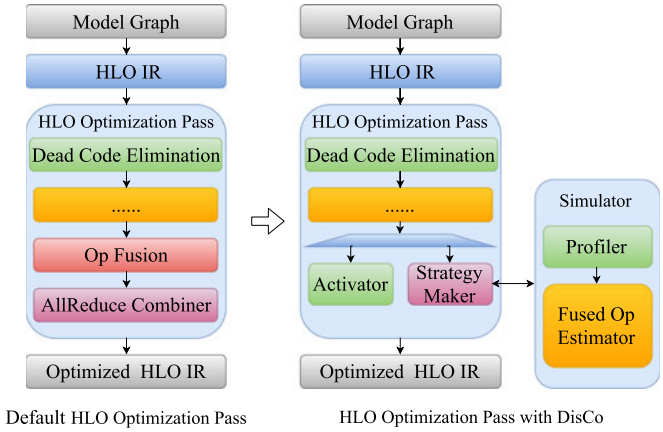


Fig. 4. Overall architecture of *DisCo*.

DisCo has two phases: *Search Phase* and *Enactment Phase*. In the *Search Phase*, the *Strategy Maker* (which runs on the master node in the TensorFlow framework) uses a backtracking algorithm to jointly search for the best op/tensor fusion strategies for distributed DNN training. In the *Enactment Phase*, the *Activator* (residing on each worker) retrieves the HLO module optimized with the best strategies to each worker, and activates the strategies for distributed training.

To facilitate the backtracking search algorithm in *Strategy Marker*, the *Simulator* estimates the per-iteration training time of the DNN model using candidate strategies that the backtracking algorithm generates. A GNN-based *Fused Op Estimator* predicts the execution time of fusion ops, to serve the *Simulator*. The *Profiler* runs the DNN model to record execution time of individual ops and prepares the training data for the GNN model of *Fused Op Estimator*.

DisCo provides a simple switch for developers to alter the phase of the system: when setting an environment variable `ENABLE_SEARCH` to 1, the search phase is activated and backtracking search is used to identify the best fusion strategies; when `ENABLE_SEARCH` is 0, enactment phase starts and distributed training is activated using the best strategies found in the search phase.

3.2 Strategy Maker

Our strategy space includes combinations of the following set of strategies: (i) fusion strategy for each computation op: no fusion, or fusing the op with a predecessor op p, $\forall p$ among the op's predecessor ops in the current HLO (which can be original op or fused op); (ii) fusion approach for a predecessor op which has multiple successors: duplicate fusion or non-duplicate fusion (Fig. 1); (iii) fusion strategy for each AllReduce instruction: no fusion, or combining the tensor with any of the neighboring original or fused gradient tensors. A neighbor gradient tensor is produced by a BP gradient computation op that is a successor or a predecessor to the op producing the current gradient tensor.

The goal is to minimize per-iteration training time of the DNN model, i.e., end-to-end execution time of the HLO module in the distributed setting (including execution time of computation ops and AllReduce instructions). The *Strategy Maker* exploits a backtracking algorithm to explore the

joint strategy space and exploits the *Simulator* to guide the search directions.

4 Strategy Framework

4.1 Activator

In the *Enactment Phase*, the *Activator* on the master node fetches the optimized HLO module generated by *Strategy Maker*, and broadcasts it to each of the other workers. The activators in other workers receive the HLO module and then execute the optimized HLO module together, i.e., carry out distributed training using the optimized strategies.

4.2 Simulator

In the *Search Phase*, the *Simulator* is used as a cost model to drive the backtracking algorithm in the *Strategy Maker*. It simulates training according to the strategies produced by the *Strategy Maker*, and estimates the per-iteration training time using profiled data from the *Profiler* for individual ops and the *Fused Op Estimator* for fused ops.

Profiler. It profiles distributed training of the given DNN model to obtain execution time of each HLO instruction and communication time of each AllReduce instruction across different devices. The execution time of each HLO instruction is recorded and indexed by its `op_code` and input shape. We build a linear regression model for communication time prediction of AllReduce instructions according to the tensor size: $T = Cx + D$, where T is the predicted execution time of the AllReduce instruction, x is the size of the gradient tensor, C reflects the bandwidth and D is the communication overhead in AllReduce instructions. Normally, AllReduce execution time is affected by multiple factors including tensor size, network topology and bandwidth, and the communication library in use. In our scenario, the time is most relevant to the tensor size as other factors are fixed. Taking ring AllReduce as an example, if the NICs work at the full-duplex mode, the communication time can be computed as $T = \frac{2(N-1)x}{B \times N}$ [42], where N is the number of devices and B is the smallest bandwidth between any device pair along the ring; T is linear with x when B and N are fixed, ensuring a simple linear regression model is accurate enough for our prediction purpose.

Fused Op Estimator. We design a GNN model to predict execution time of each fused op, which takes as input interconnectivity and features of ops to fuse (i.e., execution time of individual ops), and predicts execution time of the fused op.

4.3 GNN-Based Fused Op Estimator

Since each fused op consists of multiple original ops, a fused op can be regarded as a subgraph of the DNN model graph, whose nodes are the original ops and edges are the dependencies among them. A GNN is a nice fit for learning features of the subgraphs for fused op execution time prediction: the GNN takes the op type, input and output sizes, execution time of each original op and the data dependency among them as input features, and learns the execution time of the fused op as output; our prediction problem can be regarded as a GNN graph classification and regression job, which has been studied in the literature [43], [44], [45]. Based on the GNN, the *Simulator* can further calculate the execution time of the whole HLO module. We do not use a GNN to directly predict the execution time of the whole HLO module, since the computation time of individual ops can be profiled and the communication time of AllReduce instructions can be estimated using the linear regression model, and we can have more accurate estimation accordingly. An illustration of our GNN model is in Fig. 5.

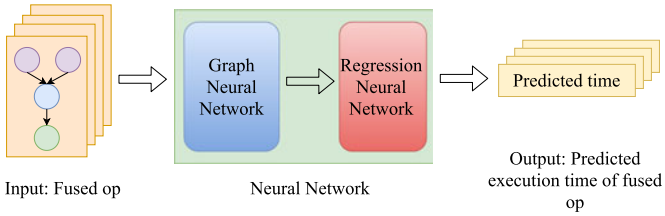


Fig. 5. GNN-based fused op estimator.

4.3.1 Feature Encoding

The GNN layers create a flat feature vector for each fused op by encoding its subgraph into a set of embeddings.

Original op Embeddings. The GNN takes as input the following subgraph information: (1) an op feature matrix, where each row corresponds to one original op and contains the op's attributes, including execution time, input and output sizes, op type (e.g., Conv2D, MatMal); (2) an adjacency matrix describing data dependencies among the ops. It generates a per-node embedding vector e_i, by encoding attributes of immediate neighbors of op i using multi-head attention layers [46]

$$\mathbf{e}_i = \|_{k=1}^{K} \sigma\left(\sum_{j \in \mathcal{N}_i} \gamma_{ij}^k W^k \mathbf{e}'_j \right). \tag{1}$$

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

Fused op Embeddings. A fused op embedding $\mathbf{y}$ is generated by encoding information from all original ops in the fused op

$$\mathbf{y} = \sigma\left(\sum_{i \in \mathcal{N}} W \mathbf{e}_i \right), \tag{2}$$

where $\mathcal{N}$ contains all the original ops in the fused op.

4.3.2 Regression Neural Network

The fused op embeddings are fed into a regression neural network [47] for execution time prediction, which consists of a number of Fully Connected (FC) layers followed by a Relu activation function.

4.3.3 Model Training

The graph neural network and regression neural network are trained together in a supervised manner, using a sampled set of fused ops, G (see Section 5.2 for details of producing GNN

training samples). For each fused op, predicted execution time is produced by the GNN model. The objective is to minimize the overall loss over the $|G|$ fused ops

$$L(\theta) = \frac{1}{|G|}\sum_{g\in G}\log\left(\mathbf{y}_g - \mathbf{y}'_g\right)^2, \tag{3}$$

where θ is the set of weights in the GNN model to learn, and $\mathbf{y}_g$ and $\mathbf{y}'_g$ are the predicted and profiled execution time of fused op g, respectively. We use Adam Optimizer [48] to minimize the loss function.

4.4 End-to-End HLO Execution Time Estimation

The simulator computes end-to-end execution time of an HLO module, by simulating scheduling process of the HLO on one device and taking AllReduce communication among this device and others into account. The complete scheduling process can be described as follows. A ready queue is maintained, consisting of computation ops whose dependencies have been cleared. Iteratively, a ready op is removed from the head of the queue, and the completion time of the op is computed according to the completion times of its predecessors and its own execution time. Then, this op's successors can be appended to the tail of the queue if the respective successor's dependencies are all cleared. AllReduce instructions are executed in order of production of their respective gradient tensors (which can be original tensor or fused tensor). An AllReduce instruction starts after its gradient tensor is produced (in case of a fused tensor, after all tensors composing the fused tensor have been produced) and the communication channel becomes clear, and its execution can overlap with the execution of computation ops in time. The simulator serves as a cost model $Cost(H)$, where H indicates the candidate HLO module, in our strategy search algorithm.

4.5 Backtracking Search

The strategy maker exploits a backtracking search algorithm to explore the joint strategy space. Algorithm 1 summarizes our search algorithm. Corresponding to the three types of strategies (Section 3.2), three optimization methods ($\mathcal{S}$) are explored in our search:

(i) Randomly choose one computation op, and fuse it with a randomly chosen predecessor op in the current HLO module; if the selected predecessor op p has multiple successor ops, redirect the output of the fused op to p's other successors. (Fig. 1 (ii)).

(ii) Randomly choose one computation op, and fuse it with a randomly chosen predecessor op in the current HLO module; if the selected predecessor op p has multiple successor ops, duplicate p and direct the output of the replica to other successors of p (Fig. 1 (iii)).

(iii) Randomly choose one AllReduce instruction, and combine it with a randomly chosen neighbor AllReduce instruction. A neighbor AllReduce instruction corresponds to a (fused) gradient tensor produced by a (fused) BP gradient computation op, neighbor to the op producing the chosen tensor.

The reasons of potentially duplicating a predecessor op (as in (ii) above) are as follows: on one hand, the time needed for re-computing the op could be smaller than data transferring time between on-chip memory and device memory when not using fusion, if the size of activations produced by the op is large; on the other hand, the output of the duplicated op can be transferred immediately to other successor ops, without waiting for completion of the fused op.

Algorithm 1. Backtracking Search

Input: input HLO module $\mathcal{H}_{in}$, optimization method set $\mathcal{S}$, cost model Cost(Â·), parameters α and β.
Output: optimized HLO module.
$\mathcal{Q} := \{\mathcal{H}_{in}\}$ # $\mathcal{Q}$ is a priority queue sorted by $Cost(\cdot)$.
$unchanged\_counter := 0$ # a counter to record the number of steps in which $\mathcal{H}_{Opt}$ has not been changed.
while $\mathcal{Q} \neq \{\}$ and $unchanged\_counter < 1000$ **do**
  $\mathcal{H} := \mathcal{Q}.dequeue()$
  for optimization method $s \in \mathcal{S}$ **do**
    # Generate a random value ranging from 0 to β.
    $n := Random(0, \beta)$
    # randomly apply s on $\mathcal{H}$ for n times.
    $\mathcal{H}' := RandomApply(\mathcal{H}, s, n)$
    if $\mathcal{H}'$ is valid **then**
      if $Cost(\mathcal{H}') < Cost(\mathcal{H}_{Opt})$ **then**
        $\mathcal{H}_{Opt} := \mathcal{H}'$
        $unchanged\_counter = 0$
      else
        $unchanged\_counter += 1$
      end if
      if $Cost(\mathcal{H}') \leq \alpha \times Cost(\mathcal{H}_{Opt})$ **then**
        $\mathcal{Q}.enqueue(\mathcal{H}')$
      end if
    end if
  end for
end while
return $\mathcal{H}_{Opt}$

To explore the strategy space for producing an optimized HLO module, a priority queue $\mathcal{Q}$ is maintained for backtracking: some candidate HLO modules, produced during the search process, are buffered in order of their $Cost()$ (i.e., end-to-end execution time) for further optimization; the optimization methods are recursively applied to these candidate HLO modules. Initially, the original HLO module is enqueued into $\mathcal{Q}$. In each search step, the algorithm dequeues the HLO module $\mathcal{H}$ from the head of $\mathcal{Q}$. Each of the three optimization methods is applied on $\mathcal{H}$ for n times (noted as $RandomApply$ in Algorithm 1), where n is a random number within the range of 0 and β (β is a positive integer). If the obtained new HLO module $\mathcal{H}'$ is valid (i.e., not including op fusion which should not be done, e.g., the op is a parameter type or control-flow op such as `switch` and `while`), we compare the execution time of $\mathcal{H}'$ with that of the best HLO module,$\mathcal{H}_{Opt}$, identified so far, and record $\mathcal{H}'$ as the best HLO module if its cost is smaller. On the other hand, if $\mathcal{H}'$'s execution time is no larger than α ($\alpha \geq 1$) times that of $\mathcal{H}_{Opt}$'s, it will be enqueued into $\mathcal{Q}$ for backtracking and optimization again in further steps. The search process continues until $\mathcal{Q}$ is empty or $\mathcal{H}_{Opt}$ remains unchanged for a number of steps (1,000), and returns the best HLO module $\mathcal{H}_{Opt}$ identified.

α and β are two key hyper-parameters in our backtracking search algorithm. β determines the probability to fuse

more ops in one step. As described in Algorithm 1, we evaluate the modified HLO module (using the simulator) once every n times rather than each time after applying an optimization method, where n is randomly generated for applying each optimization in each step with the upper bound β. This is because the change of the HLO module is subtle after only applying an optimization method once, as well as to reduce the evaluation time for expedited search. By this design, all three optimizations are randomly mixed to produce candidate HLO modules. When β is large, there is a higher probability to fuse more ops in one step. Parameter α determines pruning of the search space, since candidate HLO modules whose costs are larger than α times the cost of the best HLO module are eliminated from further exploration. Value of α decides a trade-off between the search time and performance of the best HLO module identified: a smaller α allows the search to end sooner with less recursive optimizations, while a larger value enables exploring the search space more to potentially identify better HLO modules. We will evaluate the effects of α and β in Section 6.7.

5 Implementation

DisCo is implemented based on JAX 0.2.3 [19]. JAX is an XLA-based programming framework for generating high-performance accelerator code from pure Python and Numpy ML programs. *DisCo* is implemented as a Python module that developers can readily import into their code. Core design of *DisCo* is generally applicable and can be implemented in other ML frameworks as well, as a plugin module in their graph-level optimization pass.

By default, TensorFlow XLA groups the ops in a DNN model into several clusters; it generates an individual HLO module for each cluster for further optimization passes (e.g., op fusion, common sub-expression elimination and dead code elimination), separately. This may lose the opportunity for global optimization. In single-device training, it might be still acceptable; in distributed training, joint computation and communication optimization plays an important role and has to be considered in a global view. Therefore, we build *DisCo* on JAX rather than directly based on XLA, since JAX is able to group all the core processing logic into a single HLO module for further optimization.

5.1 Activator

The activator is implemented as a module inside XLA. The activator on the master node reads the optimized HLO module from a configuration file (written by the strategy maker), and sends the HLO module to all other workers using `MPIBroadcast`.

Multi-GPU Training With JAX. Although JAX supports multi-TPU-server training, it does not support multi-machine training using GPU servers currently. To enable multi-GPU-server training with JAX, we manually modify the logic of creating the communication channels from AllReduce among GPUs on a single machine to among GPUs across multiple machines, based on NVIDIA Collective Communications Library (NCCL) [27]. In single-machine training, a unique identifer, `unique_nccl_id`, is created and used to create an AllReduce communication channel among multiple GPUs on the machine. In a multi-machine scenario, when the master node creates `unique_nccl_id`, we use `MPIBroadcast` to broadcast it to all other workers. The workers then create the inter-machine communication channel based on the global `unique_nccl_id`. One communication channel is established for AllReduce instructions among the same set of workers, using the same aggregation/reduce topology.

5.2 Strategy Maker

Profiler is implemented based on XLA's built-in profiler by adding flag `–xla_hlo_profile` to the environment variable `XLA_FLAGS`. An op 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 op.

Fused Op Estimator is implemented in Python with 2812 LoC based on Deep Graph Library (DGL) [49]. We use 6 graph convolution layers to generate original and fused op embeddings and 3 dense layers for regression. For supervised learning of the GNN model, we randomly generate different fused ops in a number of DNN models (VGG19, ResNet50, Transformer, RNNLM, BERT and Reformer). We generate 30,000 samples for each DNN model. To generate a sample, we randomly select an op and fuse it with one of its predecessors, and then repeatedly fuse this fused op with one predecessor for N times, where N is randomly selected from 1,000 to 50,000. We train the GNN model using one Tesla V100 GPU, and it takes around 14 hours till convergence. Note that this is the time to train the base GNN model from scratch. The 6 types of DNN models contain most representative types of original ops. When predicting the execution time of a fused op that contains ops not covered in these models, we fine-tune the GNN with the new op's information, which takes much less time.

6 Evaluation

6.1 Methodology

Testbed. We evaluate *DisCo* in 2 clusters. *Cluster A* consists of 6 physical machines (12 GPUs): each machine is equipped with two 11 GB NVIDIA GTX 1080 Ti GPUs, one 8-core Intel Xeon E5-1660 v4 CPU and one 100 GbE Mellanox RDMA card; all machines are connected through a 100 GbE switch.

Cluster B consists of 8 physical machines (64 GPUs): each machine is equipped with 8 16 GB NVIDIA TESLA T4 GPUs, one 96-core Intel Xeon CPU and one 100 GbE NIC.

Benchmark Models. We evaluate *DisCo* by training 2 types of CNN models (VGG19 [50], ResNet [29]) and 4 types of NLP models (Transformer [46], RNNLM [25], Bert [51] and Reformer [52]). Each model is trained using data parallelism with all GPUs in each cluster, based on produced strategies.

Baselines. We compare *DisCo* with the following baselines. (1) *JAX_no_fusion*: JAX with neither op nor AllReduce fusion; (2) *JAX_op_fusion*: JAX with XLA default heuristic op fusion, which extensively fuses ops according to a post order of ops in the DNN graph, when the ops are fusible, (this baseline represents the cases of single-device op fusion optimization combined with distributed training using AllReduce). (3) *JAX_AllReduce_fusion*: JAX with XLA default heuristic AllReduce fusion, which fuses neighboring AllReduce instructions based on a pre-defined tensor size threshold; (4) *JAX_default*:

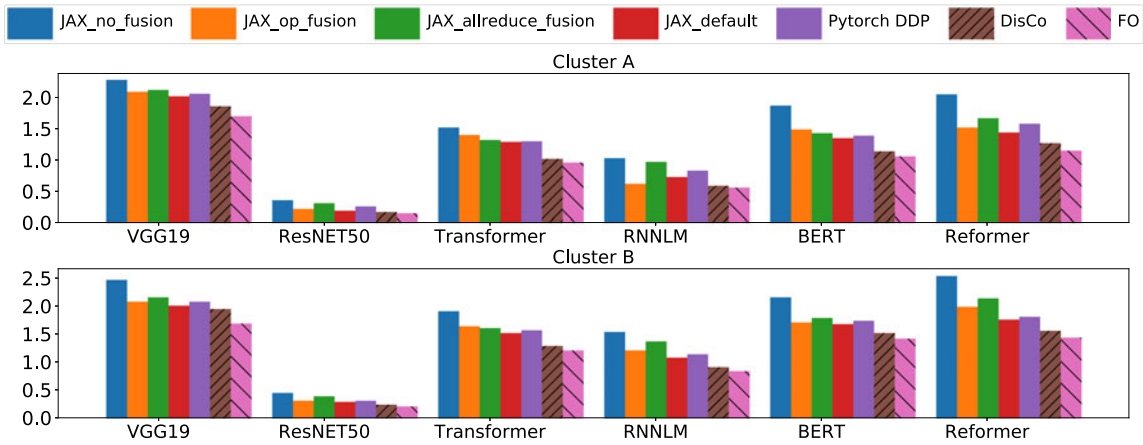


Fig. 6. Per-iteration training time comparison in Clusters A and B.

JAX with XLA default heuristic op and AllReduce fusion strategies. (5) *Pytorch DDP*: PyTorch DPP [53] overlaps AllReduce with the backward and forward passes. It does not consider op fusion.

We further compare *DisCo*'s op fusion with those in representative single-device DL compilers, TVM [1], nGraph [22] and TASO [26].

We compare with JAX instead of XLA-enabled Tensorflow [5], because JAX outperforms original XLA-enabled Tensorflow by grouping almost all ops into one cluster for global jit compilation optimization [19].

Default Setting. Unless stated otherwise, we use $\alpha = 1.05$ and $\beta = 10$ in *DisCo*'s search algorithm. To train each DNN model in a cluster, we use a batch size that can maximally exploit capacities of the respective GPU. The rationale is that if a single GPU is not fully utilized, there is no need to scale the training to many GPUs; we may just reduce the number of GPUs in use while fully utilizing each GPU.

6.2 Training Speed-Up

In Fig. 6, we compare the average per-iteration training time of different models trained on our two clusters, using strategies produced by *DisCo* and the five baselines. We observe that *DisCo* always performs the best. The fully overlapping (*FO*) execution time is given as a performance upper bound (i.e., lower bound of per-iteration training time), computed by maximally overlapping computation and communication without considering their inter-dependencies. Table 1 summarizes the speed-ups of *DisCo*, computed by $(T_{min} - T_{Disco})/T_{Disco}$, where T_{min} is the minimum per-iteration training time achieved among the baselines and T_{Disco} is the per-iteration time of *DisCo*. It also lists the speed-ups achieved in the FO cases, computed by dividing the difference of FO's per-iteration training time and the minimum time achieved among the baselines by FO's per-iteration training time. We do not include the search time when computing the training speed-up because the search is done offline and identified strategies can be used during the entire training process. Further, the search time is much smaller than the entire training time (e.g., within a few hours versus several days).

TABLE 1
Speed-Ups of *DisCo* and the FO Case Compared to the Best Performance Among the Baselines

Models	Cluster A		Cluster B	
	DisCo	FO	*DisCo*	FO
VGG19	8.6%	18.9%	10.1%	12.5%
ResNet50	12.5%	28.5%	9.6%	16.8%
Transformer	26.7%	34.7%	20.6%	25.9%
RNNLM	5.1%	10.9%	8.6%	12.3%
BERT	18.5%	27.6%	13.7%	19.5%
Reformer	13.4%	25.4%	14.5%	21.8%

6.3 Time Breakdown

Fig. 7 shows the average per-iteration training time, computation time and communication time of training 4 models in cluster A using baselines and *DisCo*. Due to computation-communication overlap, per-iteration training time is usually smaller than the sum of computation time and communication time. With *DisCo*, computation time is smaller than that of the baselines, due to *DisCo*'s selection of ops to fuse without a pre-defined order nor deterministic heuristics, that identifies better strategies within enlarged search space (JAX_default adopts heuristic fusion strategies); communication time is also reduced due to our search for good

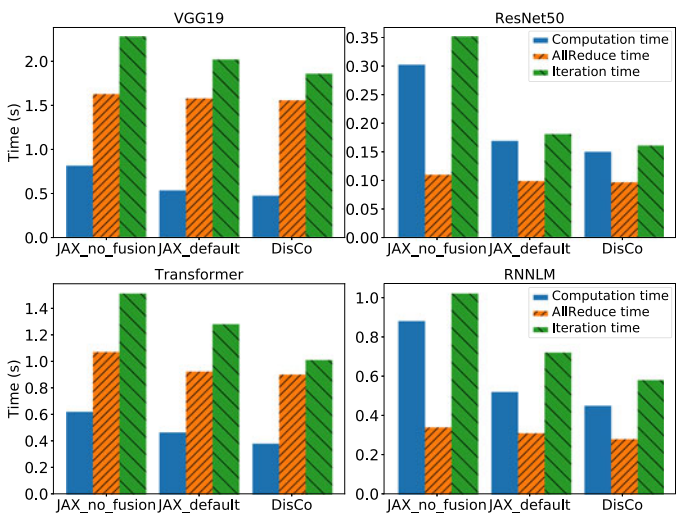


Fig. 7. Per-iteration computation/communication time.

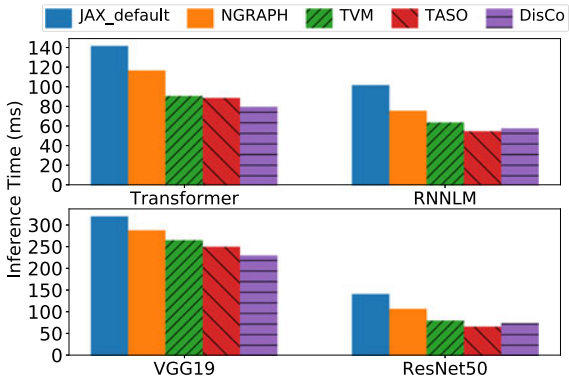


Fig. 8. Comparison of single-device inference time with representative DL compilers.

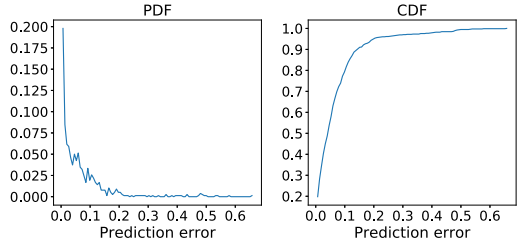


Fig. 9. Probability density function and cumulative distribution function of prediction errors of fused op estimator.

AllReduce fusion strategies (JAX_default adopts a fixed tensor size threshold to fuse AllReduce instructions).

Regarding communication-computation overlap, for the example of Transformer, the ratio of the sum of computation and communication time over per-iteration training time is 1.12 with JAX_no_fusion, 1.08 with JAX_default, and 1.27 with *DisCo*. These show that although JAX_default achieves better performance than JAX_no_fusion in terms of computation time and communication time separately, the overlap ratio drops, because its greedy op fusion delays a large amount of communication till the completion of fused ops. *DisCo* not only reduces computation time and communication time separately, but also increases the overlap ratio by jointly choosing appropriate ops and tensors to fuse. *DisCo* achieves better performance with both computation-bound models (ResNet50 and RNNLM) and communication-bound models (VGG19 and Transformer). We also notice that *DisCo* usually achieves better improvement for communication-bound models than computation-bound models. It is becuase that for computation-bound models, the main benefits come from the better fusion strategy. For communication-bound models such as Transformer, the main benefits arise from the better fusion strategy, the better AllReduce fusion strategy and the better overlapping of the communication and computation. Therefore the improvement is usually more with communication-bound models.

6.4 Single-Device Performance Comparison

Since most of the existing DL compilers focus on single-device training/inference acceleration, we also run *DisCo* on a single device (a GTX 1080 Ti GPU) to compare the model inference time achieved with *DisCo* and with representative DL compilers. JAX_default, nGraph [22] and TVM [1] use rule-based heuristics for op fusion. TASO [26] uses a search-based algorithm for graph substitution, which generates subgraph candidates and then searches for the best graph substitution. Fig. 8 shows that *DisCo* outperforms all rule-based compilers, due to identifying better op fusion strategies using the backtracking algorithm in a larger search space, while rule-based heuristics rely on the limited number of pre-defined rules. It achieves similar performance as TASO (slightly better with Transformer and VGG19, and slightly worse with RNNLM and ResNet50). *DisCo* and TASO focus on different search spaces: TASO is mainly for subgraph substitution and *DisCo* is on op fusion. The results show that in RNNLM and ResNet50, there might be more opportunities for subgraph substitution than op fusion for TASO to achieve better performance.

6.5 Simulator Accuracy

We evaluate the accuracy of our GNN-based *Fused Op Estimator* by randomly generating 2,000 unseen fused ops, which do not appear in our GNN training sample set. In this experiment, both the GNN training set and the above test samples are profiled on a GTX 1080 Ti GPU. The execution time of these fused ops ranges from 20 microseconds to 30 milliseconds. We compare the predicted execution time of fused ops in the test set and their profiled execution time, and compute a prediction error by dividing the absolute difference of these two values by the profiled execution time. Fig. 9 shows the PDF and CDF of the prediction errors. We see that more than 90% predictions are within 14% error of the respective real execution time. It shows that the GNN-based estimator can effectively learn the structural information of fused ops, by considering data and control dependencies among the original ops in the fused op subgraphs.

We then test the accuracy of the simulator for estimating the end-to-end execution time of HLO modules. Table 2 gives the estimated time by the simulator (simulation time) to execute the best HLO module found by *DisCo* on each DNN model and the respective real execution time in cluster A. The error is calculated by dividing the absolute difference of simulation time and real execution time by the real execution time. The simulator achieves a 11.1% error ratio for RNNLM and at most 17.5% for Reformer, which is good enough to guide the search algorithm. It also implies that the linear regression model for estimating the execution time of AllReduce instructions is accurate enough in spite of its simple form.

6.6 Effects of Optimization Methods

We further evaluate the design of three optimization methods in *DisCo* (Section 4.5), duplicate op fusion, non-duplicate op

TABLE 2
Estimation Error of the Simulator

Models	Real Execution Time (s)	Simulation Time (s)	Error
VGG19	1.85	2.08	12.4%
ResNet50	0.16	0.18	11.1%
Transformer	1.01	1.15	13.9%
RNNLM	0.58	0.66	13.8%
BERT	1.13	1.3	15.9%
Reformer	1.26	1.48	17.5%

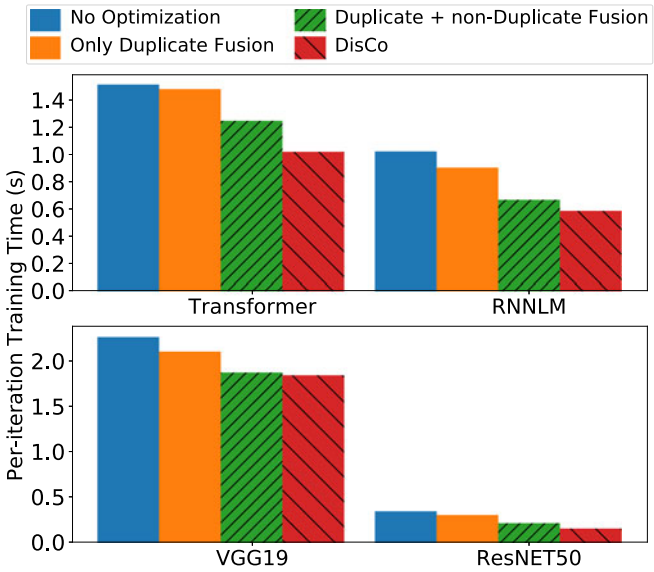


Fig. 10. *DisCo* with/without certain optimizations.

fusion and AllReduce fusion, by increasingly adding each one of them in the search algorithm, respectively, when training DNNs in cluster A. Fig. 10 shows that each optimization positively contributes to training time reduction, and the joint application of all three achieves the best performance. We further observe that non-duplicate fusion plays the most important role in reducing the training time, since per-iteration time decreases most when it is added, as compared to adding either of the other two methods. This is because compared with non-duplicate fusion, duplicate fusion leads to extra computation, and is usually suitable when the output of the predecessor op needs to be sent to other successors earlier (especially when the successor is an AllReduce instruction); in most cases, the output of a predecessor op is activation that does not need to be transferred to other devices. We also notice that in case of VGG19, *DisCo*'s performance is similar with and without AllReduce fusion. This is because most of the large gradient tensors in VGG19 are from the fully connected layers and are transferred at the beginning of back propagation; after transferring these large tensors, communication of other small tensors can overlap well with computation (with op fusion that our search algorithm identifies) even without tensor fusion.

6.7 Parameters in Backtracking Algorithm

We tune parameters α and β in our search algorithm (Algorithm 1), and train DNNs in cluster A with the respective best strategies. Tables 3 and 4 show the result per-iteration training time, along with the search time to find the respective best strategy on each DNN model.

Setting β to 10 and varying α, Table 3 shows that with a larger α, training time decreases (because the search space is larger with more candidate HLO modules enqueued and repeatedly optimized), while the search time increases accordingly. $\alpha = 1.05$ leads to a good trade-off between strategy quality and search time.

Setting α to 1.05 and varying β, Table 4 shows that when β increases, the search time decreases (because there is a higher probability to fuse more ops within one algorithm step, reducing the search space), while the training time increases in general. We observed that when β is relatively smaller (i.e., ranging from 1 to 10), the training time increases slowly but the search time drops significantly with the increase of β. When β is small, the modification of the HLO module is subtle in each step, leading to slow search progress in a huge search space. Training time increases with larger β because in each step, there is a higher probability for *DisCo* to carry out op fusion for multiple times before execution time of the produced HLO module is evaluated, which may miss part of the search space to find a better strategy. We identify $\beta = 10$ to be a good choice for the trade-off between training performance and search time.

TABLE 3
Per-Iteration Time and Strategy Search Time With Different Values of α

Models	Execution Time(s)/Search Time(min)		
	$\alpha = 1$	$\alpha = 1.05$	$\alpha = 1.1$
VGG19	2.01/11	1.85/17	1.83/36
ResNet50	0.18/15	0.16/22	0.16/48
Transformer	1.18/54	1.01/74	1.02/143
RNNLM	0.74/6	0.58/9	0.57/16
BERT	1.26/69	1.13/89	1.10/158
Reformer	1.44/53	1.26/78	1.21/161

7 Related Work

7.1 Deep Learning Compiler

MLIR [2] is a reusable and extensible compiler infrastructure that standardizes the Static Single Assignment-based IR data structures and provides a declarative system to define IR dialects. Relay [3] presents a compiler framework to unify and generalize IR in existing frameworks. Intel nGraph [22] simplifies the realization of optimized DL performance across software frameworks and hardware platforms with carefully designed IR and bridge to connect different frameworks. It carries out op fusion extensively similar to XLA's approach. TVM [1] is a compiler that exposes graph-level and operator-level optimizations to provide performance portability for DL workloads across diverse hardware backends. It defines four types of ops (injective, reduction, complex-out-fusible, and opaque), and provides generic rules to fuse these ops: multiple injective ops can be fused into another injective op; a reduction op can be fused with input injective ops; ops such as conv2d are complex-out-fusible, and their outputs can be fused

TABLE 4
Per-Iteration Training Time and Strategy Search Time With Different Values of β

Models	Execution Time(s)/Search Time(min)			
	$\beta = 1$	$\beta = 5$	$\beta = 10$	$\beta = 30$
VGG19	1.83/66	1.87/29	1.85/17	2.09/14
ResNet50	0.12/84	0.15/41	0.16/22	0.21/15
Transformer	1.00/195	1.02/98	1.01/74	1.14/54
RNNLM	0.53/64	0.54/17	0.58/9	0.76/7
BERT	1.12/258	1.14/114	1.13/89	1.26/71
Reformer	1.19/239	1.24/101	1.26/78	1.38/59

with element-wise ops. All these compilers focus on DNN training/inference on a single device.

For distributed DNN compilation, XLA integrates collective AllReduce into its HLO module; the AllReduce optimization pass and op fusion optimization pass adopt simple heuristics and are done separately. Gshard [8] is an extension of XLA which provides convenient APIs for sharding large models; no extra op fusion and AllReduce fusion strategies are provided. Boehm et al. [9] provide distributed op primitives in their customized compiler, but focus on traditional ML jobs, e.g., training KMeans [54] or L2SVM [55]. They divide a model graph into several parts and use tree search to decide the fusion strategy separately for each part; this may lose the opportunity for global optimization.

7.2 Learning-Based Prediction Model

Several learning-based cost models have been developed for automatic code optimization. MILEPOST GCC (GNU Compiler Collection) [56] uses a 1-nearest-neighbor model which takes as input manually selected features and predicts the best combinations of compiler flags for GCC. Ithemal [57] uses an LSTM model to predict the throughput of assembly-level code. Baghdadi et al. [58] integrate a DL-based cost model into an auto-scheduler, that enables the Tiramisu compiler to select the best code transformation for a given program.

In the area of DNN optimization, Kaufman et al. [47] introduce a GNN-based method for a number of optimization decisions (e.g., tile-size selection, operator fusion), based on tensor computation graphs for TPU-based training. DynaTune [59] designs a Bayesian belief model to predict the potential performance gain of each operator with uncertainty quantification, to guide the optimization process of finding better fusion strategies. We are the first in integrating a GNN-based fused op cost model for joint op and tenor fusion optimization.

7.3 Distributed Neural Network Training

Horovod [18] decouples communication from specific training frameworks and optimizes it using tensor fusion. A tensor fusion threshold `HOROVOD_FUSION_THRESHOLD` is pre-defined, and small AllReduce tensors are combined within this size threshold for transmission, which is similar to XLA's tensor fusion approach. ByteScheduler [17] advocates a priority-based tensor scheduling strategy for better communication-computation overlapping; no interaction with computation op fusion is considered. GShard [8] creates shards of weights and model states that can be split among ranks. CoCoNet [13] introduces a domain-specific language to easily express communication and computation in distributed training. Megatron-LM [10] introduces an efficient intra-layer model-parallel approach to support training of very large transformer models. GPipe [11] uses pipelining to address memory bottlenecks for training large NNs. PipeDream [12] introduces a pipelining design to overlap communication and computation for asynchronous training with convergence guarantee. These projects are focusing on model or pipeline parallelism which is orthogonal to *DisCo*.

8 Concluding Discussions

We present *DisCo*, a deep learning compiler based on JAX for distributed DNN training acceleration. *DisCo* jointly optimizes computation operator fusion and AllReduce tensor fusion using a backtracking search algorithm, maximizing the overlap of computation and communication and minimizing overall training time. A GNN-based simulator is built to effectively facilitate the search in large joint op/tensor fusion strategy space. *DisCo* achieves good training speed-up as compared with existing fusion schemes and the full communication-computation overlap case, in typical distributed environments.

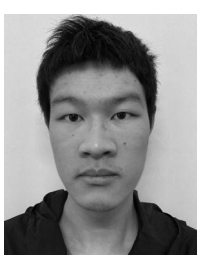
As a future direction, we plan to extend *DisCo* from data-parallel training to supporting model parallelism and pipeline parallelism. To accelerate DNN model training using model or pipeline parallelism with joint op and tensor fusion, we first need to improve our simulator: we shall profile model training on all devices and measure activation transfer time across devices as well. The optimization methods in the search algorithm should also be expanded to include fusion of activations. The HLO module will include send and recv communication instructions for activations, besides AllReduce instructions for gradient tensors. Further, the design of *DisCo* can be readily extended to handle the parameter server architecture for tensor communication, by replacing AllReduce instructions with push and pull communication instructions, while the dependencies between push and pull are readily included in the HLO module.

References

[1] T. Chen et al., "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Oper. Syst. Des. Implementation*, 2018, pp. 578–594.
[2] C. Lattner et al., "MLIR: A compiler infrastructure for the end of moore's law," 2020, *arXiv:2002.11054*.
[3] J. Roesch et al., "Relay: A high-level compiler for deep learning," 2019, *arXiv:1904.08368*.
[4] C. Leary and T. Wang, "XLA: Tensorflow, compiled," *TensorFlow Dev Summit*, 2017. [Online]. Available: https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html
[5] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Oper. Syst. Des. Implementation*, 2016, pp. 265–283.
[6] T. Chen et al., "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015, *arXiv:1512.01274*.
[7] M. Li et al., "The deep learning compiler: A comprehensive survey," 2020, *arXiv:2002.03794*.
[8] D. Lepikhin et al., "GShard: Scaling giant models with conditional computation and automatic sharding," 2020, *arXiv:2006.16668*.
[9] M. Boehm, B. Reinwald, D. Hutchison, A. V. Evfimievski, and P. Sen, "On optimizing operator fusion plans for large-scale machine learning in systemml," 2018, *arXiv:1801.00829*.
[10] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training multi-billion parameter language models using model parallelism," 2019, *arXiv:1909.08053*.
[11] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," 2018, *arXiv:1811.06965*.
[12] A. Harlap et al., "PipeDream: Fast and efficient pipeline parallel DNN training," 2018, *arXiv:1806.03377*.
[13] A. Jangda et al., "Breaking the computation and communication abstraction barrier in distributed machine learning workloads," in *Proc. 27th ACM Int. Conf. Arch. Support Program. Lang. Oper. Syst.*, 2022, pp. 402–416.

[14] L. Ma et al., "RAMMER: Enabling holistic deep learning compiler optimizations with rtasks," in *Proc. 14th USENIX Symp. Oper. Syst. Des. Implementation*, 2020, pp. 881–897.
[15] L. Zheng et al., "Ansor: Generating high-performance tensor programs for deep learning," in *Proc. 14th USENIX Symp. Oper. Syst. Des. Implementation*, 2020, pp. 863–879.
[16] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive all-reduce scheduling for expediting distributed dnn training," in *IEEE INFOCOM 2020-IEEE Conf. Comput. Commun.*, 2020, pp. 626–635.
[17] Y. Peng et al., "A generic communication scheduler for distributed DNN training acceleration," in *Proc. 27th ACM Symp. Oper. Syst. Princ.*, 2019, pp. 16–29.
[18] A. Sergeev and M. Del Balso, "Horovod: Fast and easy distributed deep learning in tensorflow," 2018, *arXiv:1802.05799*.
[19] J. Bradbury et al., "JAX: Composable transformations of python NumPy programs," 2018. [Online]. Available: http://github.com/google/jax
[20] N. Vasilache et al., "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," 2018, *arXiv:1802.04730*.
[21] N. Rotem et al., "Glow: Graph lowering compiler techniques for neural networks," 2018, *arXiv:1805.00907*.
[22] S. Cyphers et al., "Intel nGraph: An intermediate representation, compiler, and executor for deep learning," 2018, *arXiv:1801.08058*.
[23] A. Abdolrashidi, Q. Xu, S. Wang, S. Roy, and Y. Zhou, "Learning to fuse," in *Proc. Int. Conf. Neural Inf. Process. Syst. Workshop*, 2019.
[24] G. Long, J. Yang, K. Zhu, and W. Lin, "FusionStitching: Deep fusion and code generation for tensorflow computations on GPUs," 2018, *arXiv:1811.05213*.
[25] Y. Ji, G. Haffari, and J. Eisenstein, "A latent variable recurrent neural network for discourse relation language models," 2016, *arXiv:1603.01913*.
[26] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: Optimizing deep learning computation with automatic generation of graph substitutions," in *Proc. 27th ACM Symp. Oper. Syst. Princ.*, 2019, pp. 47–62.
[27] S. Jeaugey, "NCCL 2.0," GTC, 2017. [Online]. Available: https://on-demand.gputechconf.com/gtc/2017/presentation/s7155-jeaugey-nccl.pdf
[28] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *J. Parallel Distrib. Comput.*, vol. 69, pp. 117–124, 2009.
[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[30] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-XL: Attentive language models beyond a fixed-length context," 2019, *arXiv:1901.02860*.
[31] Horovod with XLA is slower than without XLA, 2018. [Online]. Available: https://github.com/horovod/horovod/issues/713
[32] Pytorch XLA is very slow on Google colab, 2020. [Online]. Available: https://github.com/pytorch/xla/issues/2247
[33] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," 2018, *arXiv:1807.05358*.
[34] X. Yi et al., "Fast training of deep learning models over multiple GPUs," in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 105–118.
[35] X. Yi et al., "Optimizing distributed training deployment in heterogeneous GPU clusters," in *Proc. 16th Int. Conf. Emerg. Netw. EXperiments Technol.*, 2020, pp. 93–107.
[36] J. Zhou et al., "Privacy-preserving graph neural network for node classification," 2020, *arXiv:2005.11903*.
[37] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" 2018, *arXiv:1810.00826*.
[38] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," 2018, *arXiv:1802.09691*.
[39] S. M. Kazemi and D. Poole, "Simple embedding for link prediction in knowledge graphs," 2018, *arXiv:1802.04868*.
[40] W. Fan et al., "Graph neural networks for social recommendation," in *Proc. World Wide Web Conf.*, 2019, pp. 417–426.
[41] S. Wu, Y. Tang, Y. Zhu, L. Wang, X. Xie, and T. Tan, "Session-based recommendation with graph neural networks," in *Proc. AAAI Conf. Artif. Intell.*, 2019, pp. 346–353.
[42] S. Kim et al., "Parallax: Sparsity-aware data parallel training of deep neural networks," in *Proc. 14th EuroSys Conf.*, 2019, Art. no. 43.
[43] F. Errica, M. Podda, D. Bacciu, and A. Micheli, "A fair comparison of graph neural networks for graph classification," 2019, *arXiv:1912.09893*.
[44] C. Cai and Y. Wang, "A simple yet effective baseline for non-attributed graph classification," 2018, *arXiv:1811.03508*.
[45] H. Li, X. Wang, Z. Zhang, and W. Zhu, "OOD-GNN: Out-of-distribution generalized graph neural network," 2021, *arXiv:2112.03806*.
[46] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 6000–6010.
[47] S. J. Kaufman et al., "A learned performance model for tensor processing units," 2020, *arXiv:2008.01040*.
[48] Z. Zhang, "Improved adam optimizer for deep neural networks," in *Proc. IEEE/ACM 26th Int. Symp. Qual. Service*, 2018, pp. 1–2.
[49] M. Wang et al., "Deep graph library: Towards efficient and scalable deep learning on graphs," *ICLR Workshop Representation Learn. Graphs Manifolds*, 2019.
[50] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[51] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.
[52] N. Kitaev, Ł. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," 2020, *arXiv:2001.04451*.
[53] S. Li et al., "Pytorch distributed," *Proc. VLDB Endowment*, vol. 13, pp. 3005–3018, 2020.
[54] K. Krishna and M. N. Murty, "Genetic K-means algorithm," *IEEE Trans. Syst., Man, Cybern., B. (Cybern.)*, vol. 29, no. 3, pp. 433–439, Jun. 1999.
[55] T. Mu and A. K. Nandi, "Breast cancer detection from FNA using SVM with different parameter tuning systems and SOM–RBF classifier," *J. Franklin Inst.*, vol. 344, pp. 285–311, 2007.
[56] G. Fursin et al., "Milepost GCC: Machine learning enabled self-tuning compiler," *Int. J. Parallel Program.*, vol. 39, pp. 296–327, 2011.
[57] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 4505–4515.
[58] R. Baghdadi et al., "A deep learning based cost model for automatic code optimization," 2021, *arXiv:2104.04955*.
[59] M. Zhang, M. Li, C. Wang, and M. Li, "DynaTune: Dynamic tensor program optimization in deep neural network compilation," in *Proc. Int. Conf. Learn. Representations*, 2020.

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.



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

Lansong Diao received the 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.

Zhen Zheng received the PhD degree from the Department of Computer Science and Technology, Tsinghua University, China, in 2019. He was a visiting scholar of North Carolina State University, in 2018. He joined Alibaba, in August 2019 as a researcher. His research interests include AI compiler, large scale machine learning systems, parallel computing and heterogeneous computing.

Shiqing Fan received the bachelor's and master's degree in computer science from Nanjing University, Jiangsu, China, in July 2015 and July 2018, respectively. He joined Alibaba Cloud since July 2018. His research interests include large-scale distributed training optimization and XLA compilation optimization.

Siyu Wang received the master's degree from 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.

Jun Yang received the master's degree from the Institute of Computing Technology, Chinese Academy of Sciences, China, in 2007. He is director of Compute Arch, NVIDIA. His research interests mainly cover machine learning system, including deep learning compiler, model compression and large-scale machine learning.

Wei Lin is currently the senior director of PAI & chief architect of big-data computation platform with 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.