Efficient Device Placement for Distributed DNN Training

Hong Wang*, Jinhao Luo*, Kaiyang Liu*, and Qiang (John) Ye§
*Department of Computer Science, Memorial University of Newfoundland, St. John's, NL, Canada
§Department of Electrical and Software Engineering, University of Calgary, Calgary, AB, Canada
Emails: hongw@mun.ca; jinhaol@mun.ca; kaiyangl@mun.ca; qiang.ye@ucalgary.ca

***Abstract*—To pursue better predictive performance in deep neural networks (DNNs), the size of learning models tends to increase, resulting in high computation requirements and training latency. In this paper, we investigate the problem of device placement to accelerate large-scale distributed DNN training. To address the challenge of finding an effective scheme due to its NP-hardness and the ever-increasing model size, we propose novel operator fusion and co-location schemes to reduce the search space while minimizing subsequent performance loss in training latency, enabling efficient device placement. We evaluate the performance of our design with real-world DNN benchmarks, and the results show that, compared to state-of-the-art approaches, our design achieves up to a 35% reduction in DNN training latency and an order-of-magnitude improvement in placement search latency.**

***Index Terms*—Distributed deep learning, device placement, operator fusion, inter-operator parallelism.**

I. Introduction

Deep neural network (DNN) models have dramatically increased in size over the past few years. For example, VGG16 [1] proposed in 2014 has over 138 million parameters, and LLAMA-3 [2] released in 2024 has an astonishing 405 billion parameters. Correspondingly, the number of training epochs and model sizes are expected to grow further to pursue better learning performance, resulting in much prolonged training delays of days, weeks, or even months for large-scale models. As modern DNN models grow in size, training on a single device becomes impractical due to time and memory constraints, necessitating distributed training across multiple devices. Model parallelism [3], a common approach for distributed training, partitions model parameters into disjoint subsets, with each device processing a specific portion of the model. It can be categorized into pipeline parallelism [4], tensor parallelism [5], and inter-operator parallelism [6]. Compared with pipeline parallelism, which performs model partitioning at the layer level, and tensor parallelism, which splits individual operators across devices, inter-operator parallelism maximizes parallelization opportunities for operators within a DNN layer that have no inter-dependencies, while inducing less communication cost. An operator is a mathematical computational unit that performs tasks such as matrix addition and multiplication, while a device represents a form of computation resource, such as a GPU, a CPU, or a tensor processing unit (TPU).

In inter-operator parallelism, a DNN computation graph is partitioned into disjoint subgraphs. Optimizing device placement and operator scheduling minimizes training latency [7].

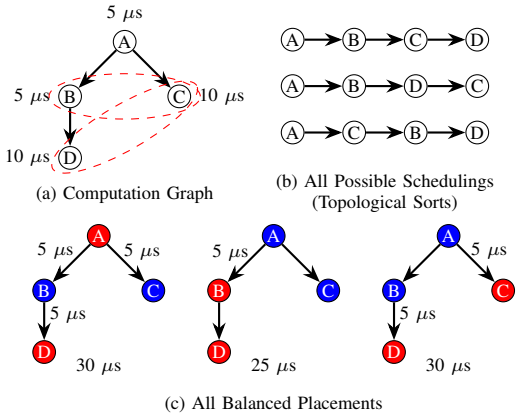


Fig. 1. Each red-dashed ellipse in (a) represents a pair of parallelizable operators. Multiple topological sorts in (b) are possible due to these parallelizable operator pairs. In (c), red and blue nodes indicate tasks running on GPU 1 and 2, respectively. We assume each cross-GPU communication takes five microseconds. The number below each placement represents the training latency under the optimal scheduling.

Placement establishes a one-to-one mapping between devices and subgraphs, while scheduling defines the sequential execution order of operators within each subgraph. An effective placement parallelizes operators that have no inter-dependencies, enabling more tasks to be processed simultaneously. Similarly, efficient scheduling reduces device idle time by minimizing delays in waiting for data from other devices.

A DNN computation graph can often be structured as a directed acyclic graph (DAG), where each node represents an operator, and each directed edge indicates data dependencies between two operators. For a pair of operators with inter-dependency, such as (A, B) in Fig. 1(a), the dependent operator B cannot begin execution until operator A is completed. Fig. 1(a) also uses red-dashed ellipses to indicate parallelizable operator pairs with no directed path connecting them. For the unique pairs of parallelizable operators, (B, C) and (C, D), the first operator in each pair can execute either before or after the second one, generating three feasible topological sorts, shown in Fig. 1(b). Any valid scheduling aligns with a topological sort that defines a linear ordering of operators such that for every directed edge from one operator to another, the first operator precedes the second one in the ordering. The presence of a topological sort is necessary and sufficient for the graph to be a DAG. Fig. 1(c) shows all balanced

placement schemes on two GPUs. The middle one in Fig. 1(c) illustrates the optimal placement where the nodes in each parallelizable operator pair, (B, C) and (C, D), are on different GPUs to enable parallel execution, leading to the shortest per-iteration training latency of 25 μs. This example demonstrates that device placement significantly impacts distributed training performance, which can often be formulated as an integer programming (IP) problem with NP-hard properties [7]. The search space for an optimal device placement expands rapidly with the scale of DNN models and the number of devices.

In this paper, we present novel operator fusion and co-location schemes to reduce the search space significantly, leading to an efficient inter-operator-level device placement. Moreover, a profiler is designed to output the operator computing latency and the memory cost, serving as inputs of the IP problem to evaluate the performance of placement and scheduling decisions[1]. Evaluation results on four real-world DNN benchmarks show that our solution can achieve up to a 35% reduction in per-iteration training latency and an improvement of over 2,000 times in placement search latency.

II. Related Work

Heuristic solutions: Existing graph partitioning tools, such as Metis [8], divide computation graphs into load-balanced subgraphs while minimizing cross-device data transmission. Additionally, starting from a random solution, local search [8] greedily improves the best single-operator placement reassignment repeatedly until a local optimum is reached. In contrast, Markov Chain Monte Carlo (MCMC) [5], [9] iteratively proposes a new strategy by randomly changing the parallelization configuration of an operator, rapidly exploring the search space. These heuristic-based methods are efficient but without the worst-case performance guarantee.

Learning-based solutions: Numerous learning-based strategies are developed to improve device placement and scheduling. For example, HeteroG [10] and TAG [11] utilize graph neural networks (GNNs) to extract computation graph information and produce generalizable device placement policies. However, these methods require extensive training time for GNNs to achieve decisions that outperform existing heuristics.

Solver-based solutions: Generally, solver-based solutions introduce high latency when searching for optimal placements in large-scale, NP-hard device placement problems. For example, Pesto, proposed in [7], models device placement and scheduling as an integer linear programming problem and applies a graph coarsening technique to accelerate problem solving. However, jointly solving device placement and scheduling problems incurs significant search latency, as both problems are NP-hard. Additionally, aggressive batch merging reduces parallelization opportunities and causes performance loss. In [8], Tarnawski *et al.* formulate an integer programming (IP) problem for placement search and enhance parallelism by introducing non-contiguous cuts, which is effective for small-scale problems without network heterogeneity.

In summary, heuristic schemes suffer from high performance loss in preserving low training latency, while learning-based and solver-based schemes face high computational complexity.

III. System Model and Problem Formulation

A. Deep Learning Clusters

We consider a homogeneous scenario where each server is equipped with k identical GPUs[2] for distributed DNN training. The computing delay of each operator is uniform across all GPUs. Because different learning tasks may have varying resource and security requirements, which can cause significant and unpredictable performance degradation when running in parallel [12], we assume that each GPU can be assigned to only one learning task at any given time. Then, the computing system scales from individual servers to clusters. Servers are interconnected via electrical packet switches arranged in a multi-tier Fat-Tree network topology [13], a hierarchical and scalable network structure consisting of three layers of Ethernet switches, widely used in modern data centers and high-performance computing (HPC) environments.

We consider a Fat-Tree topology that provides equal bandwidth across all links connecting servers, enabling all servers to communicate at the full bandwidth of b_{en}. Within each server, the GPUs are connected via NVLink, enabling direct and high-speed connections between GPUs with a bandwidth of b_{nv}. In modern HPC clusters, each server is equipped with multiple network interface cards (NICs), each supporting various ports. Since modern NICs support full-duplex capability and each port bandwidth is independent, multiple data transfers can occur simultaneously, reducing GPU idle time (caused by waiting for dependencies) and increasing overall training throughput.

B. Distributed Deep Learning and Device Placement

Computation graph: The DNN computation graph is represented as a DAG, denoted by $G = (V, E)$, where each node $i \in V$ denotes an operator such as convolution or matrix multiplication. Meanwhile, a directed edge $(i, j) \in E$ indicates that the operator j requires the output of i for processing.

Device topology: A directed graph $T = (D, L)$ represents the device topology, where each device $d \in D$ is a GPU with maximum memory capacity M_d, and each edge $(d, e) \in L$ is a communication link from d to e with bandwidth $b_{d,e}$.

Distributed deep learning: In inter-operator parallelism, operators are distributed across devices. Each device processes its assigned operators on a mini-batch in parallel with others.

Placement: Based on the input computation graph G and device topology T, we define a placement strategy $\mathcal{P}$ as $\mathcal{P} = \{(g_d, d) \mid d \in D,\ g_d \in \mathcal{G}\}$ where each pair (g_d, d) represents a unique one-to-one mapping between each GPU $d \in D$ and a corresponding subgraph $g_d \in \mathcal{G}$, with $\mathcal{G}$ representing the set of all disjoint subgraphs of G after partition.

After the profiler calculates the computing delay p_i and memory consumption m_i for each operator $i \in V$, we formulate the

[1]The implementation of our proposed solution is publicly available at https://github.com/MUN-DML/QuickP for further reference and use.

[2]"GPU" and "device" are used interchangeably throughout the rest of the paper.

device placement search as an IP problem, where A is the finish time of the last completed operator, that is, $\max_{i \in V} f_i$. By solving (P1), we aim to find an optimal placement strategy $\mathcal{P}^*$ to minimize the per-iteration latency A for a given computation graph G and device topology T. We utilize Kahn's algorithm [14] to obtain a topological sort of the computation graph G. The algorithm produces an ordered list $TP = (i_1, i_2, \ldots, i_n)$, where $\{i_1, i_2, \ldots, i_n\} = V$. A topological order tp_i is assigned to each operator $i \in V$, where tp_i corresponds to the index of operator i in TP. $O = \{(i,j) \mid tp_i < tp_j,\ i,j \in V\}$ lists all unique operator pairs (i,j) where the first operator i has a lower topological order than the second operator j, such that $tp_i < tp_j$. This avoids considering repeated operator pairs and maintains valid scheduling within each subgraph $g_d \in \mathcal{G}$. The mathematical formulation of (P1) is shown below.

$$\text{(P1):} \min_{\{x_{i,d}\}} \quad A = \max_{i \in V} f_i$$

s.t.

$$x_{i,d} \in \{0,1\},\ \forall i \in V,\ \forall d \in D, \tag{1}$$

$$M_d \geq \sum_{i \in V} m_i \cdot x_{i,d},\ \forall d \in D, \tag{2}$$

$$\sum_{d \in D} x_{i,d} = 1,\ \forall i \in V, \tag{3}$$

$$s_i + p_i = f_i,\ \forall i \in V, \tag{4}$$

$$\begin{cases} f_i \leq s_j, & \text{if } t_{i,j} = 0, \\ f_i + \sum\limits_{\substack{d,e \in D \\ d \neq e}} x_{i,d} \cdot x_{j,e} \cdot \frac{t_{i,j}}{b_{d,e}} \leq s_j, & \text{else}, \end{cases} \quad \forall (i,j) \in E, \tag{5}$$

$$f_i \leq s_j + Q \cdot (2 - x_{i,d} - x_{j,d}),$$
$$\forall (i,j) \in O,\ \forall d \in D,\ p_i > 0,\ p_j > 0. \tag{6}$$

In (P1), $x_{i,d}$ in constraint (1) is a binary variable that denotes the mapping between operator i and device d. Specifically, $x_{i,d} = 1$ if operator i is allocated to device d, and $x_{i,d} = 0$ otherwise. Constraint (2) limits the total memory of assigned operators to the device's memory capacity M_d of device d. Constraint (3) ensures each operator is allocated to one device, forming a many-to-one mapping between operators and devices. Constraint (4) indicates that for any operator i, its complete time f_i is the sum of its start time s_i and its computing delay p_i. The profiler evaluates the computing delay p_i for each operator i. In a homogeneous GPU environment, the computing delay of each operator remains uniform across all devices. Constraint (5) maintains the global data dependency, ensuring that if there is an edge $(i,j) \in E$, operator j starts execution only after i is completed and the corresponding tensor transmission is received. This transmission may incur communication costs, which are calculated based on three possible scenarios:

- For `PlaceHolder` operators, the output tensor size $t_{i,j}$ is zero, eliminating communication costs regardless of the placement of the operators i and j.
- If both operators i and j are placed on the same device, there is no communication cost. In this case, $f_i \leq s_j$.
- If operators i and j are placed on different devices (i.e., $d \neq e$), the cost is equal to the transmission tensor size $t_{i,j}$ divided by the bandwidth $b_{d,e}$ of the corresponding communication link, considering network heterogeneity.

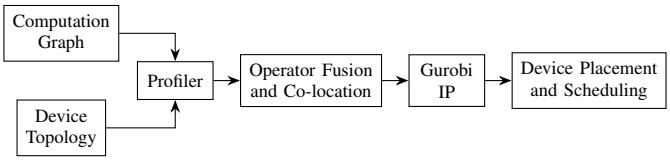


Fig. 2. Our solution consists of a profiler to evaluate the computing and memory costs of operators, along with an IP model that employs operator fusion and co-location schemes for efficient placement.

The constraint (6) indicates that for each unique operator pair $(i,j) \in O$, the operator j can only execute after i is completed if they are on the same device. This same-device condition is formulated as $Q \cdot (2 - x_{i,d} - x_{j,d}) = 0$, where Q is a sufficiently large constant. This constraint guarantees non-overlapping computing periods and valid scheduling, complying with the topological sort, on each device. Furthermore, the conditions $p_i > 0$ and $p_j > 0$ simplify the model, since operators such as `Reshape`, `ReadVariable`, `PlaceHolder`, and `Cast` in a TensorFlow graph have zero computing delay. If either operator i or j has zero computing delay, their computing periods do not overlap.

IV. Solution Design

Since finding the optimal parallelization strategy alone is an NP-complete problem [5], and considering that network heterogeneity further complicates this problem, finding an optimal solution to the device placement problem (P1) within polynomial time is infeasible. Typically, the placement search latency of commercial solvers increases exponentially with the problem size. Therefore, we propose novel operator fusion and co-location schemes to reduce the search space of (P1) while minimizing the subsequent performance loss in training latency, significantly decreasing placement search latency. Fig. 2 shows the workflow of our proposed solution framework.

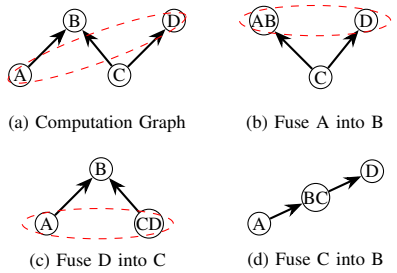
A. Parallelizable Operator Pairs

Definition 1. *In a DAG $G = (V, E)$, an operator pair (i,j), where $i,j \in V$, is parallelizable if there is no $i \rightsquigarrow j$ or $j \rightsquigarrow i$ where $\rightsquigarrow$ represents a directed path between two operators.*

To minimize the number of operator pairs involved in the IP, we only consider parallelizable operator pairs. The topological sort rule states that if there is a directed path $i \rightsquigarrow j$, the operator i must appear before j in the topological sort for any DAG. Thus, we exclude any operator pair connected by a directed path, reducing the computing complexity of the constraint (6), as the constraint (5) guarantees the global data dependency.

B. Operator Fusion

The DNN computing graph generated by TensorFlow often includes thousands or even tens of thousands of operators. We aim to fuse operators to reduce their number, accelerating the solver-searching process. One challenge of operator fusion is to prevent cycle creation, as it violates the rules of a DAG. In



(a) Computation Graph (b) Fuse A into B

(c) Fuse D into C (d) Fuse C into B

Fig. 3. The red-dashed ellipse represents a pair of parallelizable operators. Merging an edge where the source node's out-degree and the destination node's in-degree are both at least two reduces parallelization opportunities.

a DAG $G = (V, E)$, the operation that merges operator j into i for each edge $(i, j) \in E$ preserves acyclicity if and only if there exists exactly one internally vertex-disjoint path from i to j in G. This result is formally established in Pesto [7].

Theorem 1. *Assume a homogeneous environment where each device has sufficient memory and every operator incurs the same computational delay across all devices in a DAG G. If there are no parallelizable operator pairs, the optimal placement is to assign all $i \in V$ to one device, given that any edge cut $(i, j) \in EC$ has a communication cost $c_{ij} \geq 0$. $EC = \{(i, j) \in E \mid (i, j) \notin \bigcup_{g_d \in \mathcal{G}} E(g_d)\}$ indicates the set of all edge cuts during graph partition, where $E(g_d)$ is the edge set of subgraph $g_d \in \mathcal{G}$.*

Proof. In a DAG $G = (V, E)$ without any parallelizable operator pairs, there exists either a directed path $i \rightsquigarrow j$ or $j \rightsquigarrow i$ for each operator pair $(i, j), \forall i, j \in V$. Thus, there is only one topological sort for G, indicating one possible sequential execution order $\{i_1, \dots i_n\}$ among all $i \in V$, where $|V| = n$. Suppose the optimal placement is not to place all nodes on the same device. Then, EC is not empty, and the training latency is $\sum_{i \in V} p_i + \sum_{(i,j) \in EC} c_{ij}$. However, the training latency is $\sum_{i \in V} p_i$ if all operators are on the same device, leading to the minimum training latency and concluding the proof. □

Theorem 2. *In a DAG $G = (V, E)$, if an edge $(i, j) \in E$ creates a cycle after fusion, the out-degree of the source node i and the in-degree of the destination node j must both be at least two.*

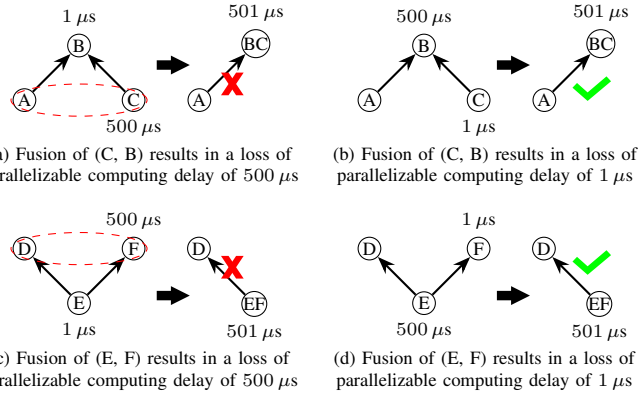
Proof. For an edge $(i, j) \in E$, if the fusion of j into i creates a cycle, there must be two or more internally vertex-disjoint paths from i to j before the fusion. However, if the out-degree of i or the in-degree of j is one, the edge (i, j) is the only path from i to j, and no cycle can be created. □

While operator fusion reduces the number of operators, it also limits opportunities for parallelization. Since it changes the graph structure, all parallelizable pairs involving the merged node become unavailable. Thus, to minimize the performance loss caused by operator fusion, we design Algorithm 1 to determine whether operator j should fuse into i for each edge $(i, j) \in E$, based on the following conditions.

Algorithm 1 Operator fusion

Input: Edge (i, j); computation graph G; merging threshold α;
Output: Boolean indicating if j will merge into i;

if $G.out\_degree(i) \geq 2 \wedge G.in\_degree(j) \geq 2$ **then**
  return False
end if
if $G.out\_degree(i) = 1 \wedge G.in\_degree(j) = 1$ **then**
  return True
end if
if $(p_i \leq \alpha \wedge G.out\_degree(i) = 1) \vee$
$(p_j \leq \alpha \wedge G.in\_degree(j) = 1)$ **then**
  return True
end if
return False



(a) Fusion of (C, B) results in a loss of parallelizable computing delay of 500 μs

(b) Fusion of (C, B) results in a loss of parallelizable computing delay of 1 μs

(c) Fusion of (E, F) results in a loss of parallelizable computing delay of 500 μs

(d) Fusion of (E, F) results in a loss of parallelizable computing delay of 1 μs

Fig. 4. Each red-dashed ellipse represents a pair of parallelizable operators. (a) and (c) show that fusing an edge may cause significant performance loss if the destination operator's in-degree or the source operator's out-degree is at least two, even if the operator has low computing delay.

- According to Theorem 2, the out-degree of operator i and the in-degree of operator j both being at least two is a necessary condition for the merge of edge (i, j) to result in a cycle. Moreover, as shown in Fig. 3(d), the fusion of operator C into B forms a straight line, eliminating all parallelization opportunities. Additionally, edges (C,B) and (C, D) are removed while a new edge (B,D) is generated, causing high fusion runtime. Therefore, skipping the fusion of such edges prevents cycle creation, reduces the fusion runtime, and preserves low training latency.
- In Fig. 1(a), edge (B, D) forms a straight line-like structure because the out-degree of B and in-degree of D are both one. By Theorem 1, the straight line B $\rightarrow$ D only has one possible topological sort, so assigning both operators to the same device is optimal. Thus, operator D can be fused into B without any performance degradation, while maintaining opportunities for parallelization with C.
- Analysis of profiling results shows that most computing demand comes from a few operators, aligning with the findings of Pesto [7]. We define α as the merging threshold, a parameter that balances placement search latency and training latency. As α increases, search latency

decreases, while training latency increases. To minimize performance degradation, an edge (i, j) can be fused if the computing delay of j is below α and its in-degree is one, or if the computing delay of i is below α and its out-degree is one, as shown in Fig. 4.

C. Operator Co-location

When operator fusion reduces the number of operators, applying operator co-location constraints further narrows the search space. Unlike fusion, operator co-location does not alter graph structure or generate cycles. We implement critical path-based heuristics (7) to calculate rank r_i of each operator $i \in V$. The calculation follows a reverse topological sort, where r_i indicates the cumulative computing and communication cost of the longest path starting from operator i, given by

$$r_i = p_i + \max_{j \in \text{succ}(i)} (r_j + \frac{t_{i,j}}{b}),\ b = \begin{cases} b_{nv}, & \text{if } |\mathcal{D}| \leq k, \\ b_{en}, & \text{else.} \end{cases} \quad (7)$$

Then, each operator i with more than one immediate successor co-places with the immediate successor j with the highest combined cost of r_j and the corresponding communication cost, that is, $\max_{j \in succ(i)}(r_j + \frac{t_{i,j}}{b_{en}})$, where $succ(i)$ represents the set of immediate successors of operator i. Afterward, the edge (i, j) is added to the edge set N.

After iterating through all operators i with more than one immediate successor, that is, $|succ(i)| > 1$, all edges $(i, j) \in N$ form a subgraph of the computation graph G, which consists of a set of weakly connected components (WCCs), denoted as W. The operators in each WCC constitute a co-location group $U \in W$, indicating that all the operators in U are on the same device. Additionally, each operator belongs to at most one co-location group $U \in W$.

IP reformulation: After applying operator co-location constraints, the reformulated IP with a significantly reduced search space is presented in (P2).

$$\text{(P2):} \min_{\{x_{i,d}\},\{y_{U,d}\}} A = \max_{i \in V} f_i$$

s.t.

Constraints (1), (2), (4),

$$y_{U,d} \in \{0, 1\},\ \forall U \in W,\ \forall d \in D, \quad (8)$$

$$\sum_{d \in D} y_{U,d} = 1,\ \forall U \in W, \quad (9)$$

$$x_{i,d} = y_{U,d},\ \forall d \in D,\ \forall U \in W,\ \forall i \in U, \quad (10)$$

$$\sum_{d \in D} x_{i,d} = 1,\ i \notin \bigcup_{U \in W} U, \quad (11)$$

$$\begin{cases} f_i \leq s_j, & \text{if } t_{i,j} = 0 \vee (\exists U \in W,\ \{i, j\} \subseteq U), \\ f_i + \sum\limits_{\substack{d,e \in D \\ d \neq e}} x_{i,d} \cdot x_{j,e} \cdot \frac{t_{i,j}}{b_{d,e}} \leq s_j, & \text{else,} \end{cases} \quad \forall (i, j) \in E, \quad (12)$$

$$\begin{cases} f_i \leq s_j, & \text{if } \exists U \in W,\ \{i, j\} \subseteq U, \\ f_i \leq s_j + Q \cdot (2 - x_{i,d} - x_{j,d}),\ \forall d \in D, & \text{else,} \end{cases}$$
$$\forall (i, j) \in R,\ p_i > 0,\ p_j > 0, \quad (13)$$

where $y_{U,d}$ represents co-location group-device mapping indicators, and constraint (9) enforces a many-to-one relationship between groups and devices. If a group U is placed on a device, all operators $i \in U$ are assigned to this device accordingly, as in the constraint (10). The many-to-one operator-to-device mapping applies only to ungrouped operators to reduce the complexity, as specified in the constraint (11). The operator i is ungrouped if it is not co-located with any other operator, that is, $i \notin \bigcup_{U \in W} U$. Unlike constraint (5), constraint (12) specifies that operators in the same group U do not incur communication costs because they are on the same device. $R = \{(i, j) \in O \mid\ i \not\rightsquigarrow j\}$ represents all parallelizable operator pairs in G. By restricting the constraint (13) to R, its complexity is significantly reduced.

The commercial solver, i.e., Gurobi, optimizes the IP (P2), producing device placement and scheduling. Then, the per-iteration training latency is obtained.

V. Performance Evaluation

A. Implementation

The proposed solution is implemented using TensorFlow 2.16 [15]. The `tf.function` decorator is applied to convert graphs from the model training iteration, since TensorFlow stores DNNs as DAGs. During training, the `graph` API is used to trace and extract the computation graph, which is then modeled using NetworkX [16]. Each node is annotated with the operator name, inputs, and outputs, while each directed edge is labeled with the transmitted tensor size. During profiling, we train the DNN model for 50 iterations, excluding the first five warm-up ones. Given that large-scale DNN training requires over 100,000 iterations and multiple epochs, this incurs a profiling overhead of less than 0.1% [7]. The Tensorflow profiler calculates the computing delay and memory cost of the operators, which serve as input to the IP model implemented by the commercial solver Gurobi [17].

B. DNN Models and Datasets

We train image classifiers, including AlexNet [18] and VGG16 [1], using the CIFAR-10 [19] dataset of ten image classes with a batch size of 512. For natural language processing (NLP) models, including BERT [20] and FNet [21], we use IMDB reviews of two classes from the TensorFlow dataset with a batch size of 16. Each input is padded to a length of 128 to ensure uniform text length.

C. Experimental Setup

We conduct profiling on the NVIDIA RTX-3070 GPU. We assume a cluster where each server has two GPUs connected via NVLink, with a bandwidth of 50 GB/s, while inter-server communications over Ethernet are at 20 GB/s. The merging threshold, α, for each model is set to the 90th percentile value in the ascendingly sorted list of non-zero operator computing delays. The IP optimizer, Gurobi, operates on an Intel Core i7-12700K CPU and outputs the placement once it guarantees a solution within 5% of the optimum. For each model, the optimization process is executed 20 times using the same

TABLE I
AVERAGE PLACEMENT SEARCH LATENCY COMPARISON IN SECONDS

Model	# of Operators (Pre → Post Fusion)	# of WCC Groups	Operator Fusion Algo. Runtime	Solver Runtime of Our Solution			Metis (All # of GPUs)
				Two GPUs	Four GPUs	Six GPUs	
AlexNet	$1,624 \rightarrow 264$	47	0.1	1.3	4.2	12.0	0.2
VGG-16	$3,598 \rightarrow 509$	124	0.1	0.5	1.9	3.7	0.6
FNet	$7,145 \rightarrow 1081$	242	0.3	3.2	5.2	16.2	1.6
BERT	$12,566 \rightarrow 2048$	429	1.0	2.5	7.1	18.9	4.2

number of devices. Then, we average the placement search latency and expected training latency.

D. Baseline

Metis [8]: Metis is a publicly available graph partitioning tool. We use computing delay as the operator weight and output tensor size as the edge weight. Metis balances the sum of operator weights across subgraphs while minimizing the total edge cut weight, ensuring load balancing and reducing communication costs.

MCMC [5], [9]: MCMC maintains a current placement strategy and iteratively proposes a new one by randomly changing the assigned device of an operator. The new strategy replaces the current one if it has a lower cost evaluation from the cost model, rapidly exploring the large search space.

Given a consistent computation graph and device topology, each baseline strategy determines a device placement, which we evaluate using the list scheduling scheme from HeteroG [10].

E. Result Analysis

Placement search latency: Table I compares the placement search latency of our solution with other baseline strategies. For our solution, placement search latency generally increases with the number of operators and devices. In contrast, the placement search latency of Metis remains constant across different GPU counts, as it relies on multilevel coarsening to perform the initial K-way cut on a small graph.

In Fig. 5, MCMC performance in achieving low training latency improves with the number of steps but shows diminishing returns over time. While MCMC is effective on models with small search spaces, the time and number of steps to find an optimal placement grow exponentially as the search space expands. When training AlexNet, MCMC requires 50 minutes to achieve a similar performance to our solution. In contrast, MCMC fails to find an effective placement with performance comparable to our design for FNet and BERT, even after 12 and 30 hours, respectively.

Per-iteration training latency: The simulation results in Fig. 6 for four DNN models show that our solution consistently outperforms all baseline strategies and achieves up to a 35% reduction in training latency when training across various GPU numbers. In contrast, Metis considers only load balancing and communication cost minimization without accounting for parallelization opportunities. As the number of devices increases,

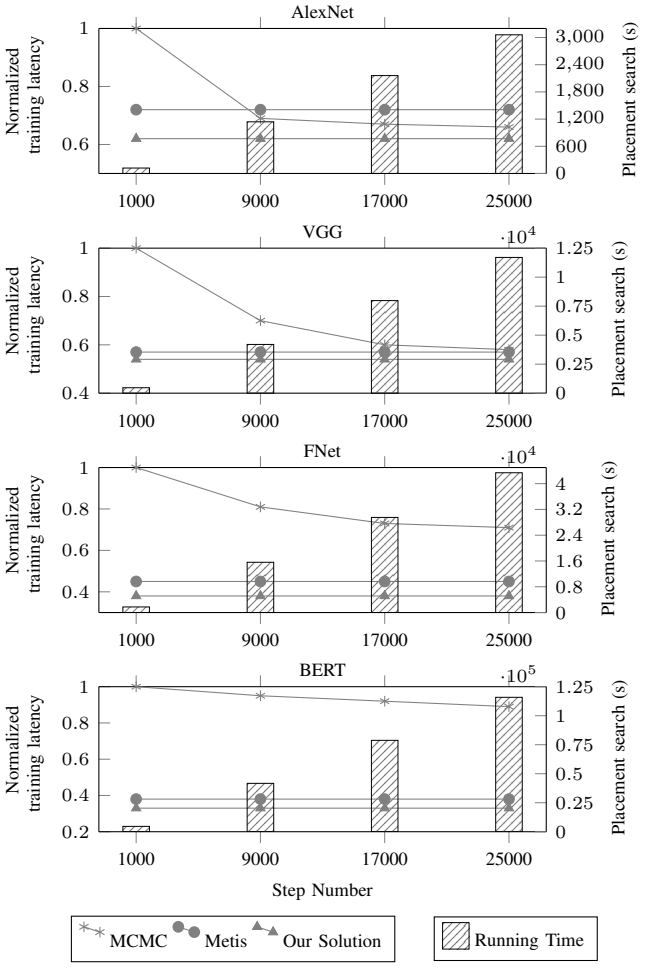


Fig. 5. This figure compares normalized training latency and placement search latency across different step numbers when training various DNN models on two GPUs. Line charts correspond to the normalized training latency of each strategy, while bars denote the placement search latency of MCMC.

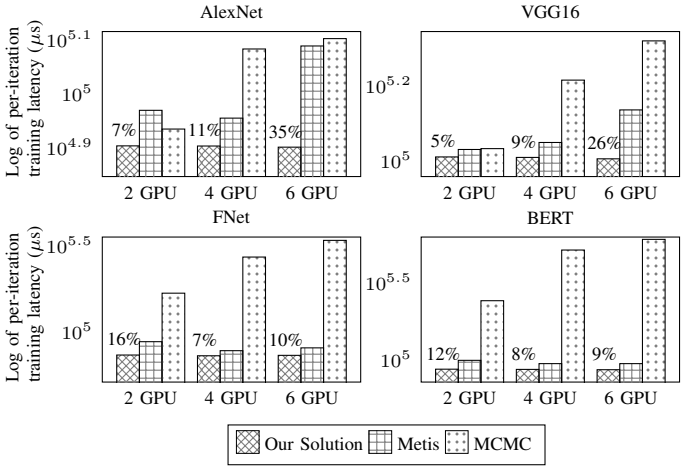


Fig. 6. This figure shows a comparison of per-iteration training latency on a logarithmic scale across all strategies. The number above each column indicates the performance improvement of our solution over the best alternative strategy. MCMC per-iteration training latency is recorded after 25,000 steps.

subgraphs on different devices tend to develop stronger inter-dependencies, leading to device idle time while waiting for cross-device dependencies and ultimately limiting scalability.

VI. Conclusions and Future Works

In this paper, we investigate the problem of device placement to accelerate large-scale distributed DNN training, tackling the challenges of NP-hardness and increasing DNN model sizes. Specifically, we present a novel approach that leverages operator fusion and co-location to reduce the search space and minimize subsequent performance loss in training latency, enabling efficient inter-operator-level device placement. Our method merges operators without inter-parallelization opportunities and co-locates those contributing to the highest cumulative computing and communication costs. Simulation results on real-world DNN benchmarks show that our solution achieves up to a 35% reduction in per-iteration training latency and an improvement of more than 2,000 times in search latency.

In future work, this approach can be adapted to a heterogeneous environment where each operator experiences varying computing delays on different devices. Furthermore, we plan to extend our solution to support data and tensor parallelism by duplicating or splitting operators.

References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[2] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, and A. Letman, "The Llama 3 herd of models," 2024.

[3] S. Moreno-Alvarez, J. M. Haut, M. E. Paoletti, and J. A. Rico-Gallego, "Heterogeneous model parallelism for deep neural networks," *Neurocomputing*, vol. 441, pp. 1–12, 2021.

[4] J. Liu, J. H. Wang, C. Rong, and J. Wang, "Pipecompress: Accelerating pipelined communication for distributed deep learning," in *IEEE ICC*, 2022, pp. 207–212.

[5] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," in *MLSys*, vol. 1, 2019, pp. 1–13.

[6] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, and Y. Huang, "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," in *USENIX OSDI*, 2022, pp. 559–578.

[7] U. U. Hafeez, X. Sun, A. Gandhi, and Z. Liu, "Towards optimal placement and scheduling of DNN operations with pesto," in *Middleware*, 2021, pp. 39–51.

[8] J. M. Tarnawski, A. Phanishayee, N. Devanur, D. Mahajan, and F. N. Paravecino, "Efficient algorithms for device placement of DNN graph operators," in *NeurIPS*, vol. 33, 2020, pp. 15 451–15 463.

[9] W. Wang, M. Khazraee, Z. Zhong, M. Ghobadi, Z. Jia, and D. Mudigere, "TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs," in *USENIX NSDI*, 2023, pp. 739–767.

[10] X. Yi, S. Zhang, Z. Luo, G. Long, L. Diao, and C. Wu, "Optimizing distributed training deployment in heterogeneous GPU clusters," in *ACM CoNEXT*, 2020, pp. 93–107.

[11] S. Zhang, X. Yi, L. Diao, C. Wu, S. Wang, and W. Lin, "Expediting distributed DNN training with device topology-aware graph deployment," *IEEE TPDS*, vol. 34, no. 4, pp. 1281–1293, 2023.

[12] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-Aware cluster scheduling policies for deep learning workloads," in *USENIX OSDI*, 2020, pp. 481–498.

[13] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Trans. Comput.*, vol. C-34, pp. 892–901, 1985.

[14] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962.

[15] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, and J. Dean, "Tensorflow: A system for large-scale machine learning," in *USENIX OSDI*, 2016, pp. 265–283.

[16] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *SciPy*, 2008, pp. 11–15.

[17] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2024. [Online]. Available: https://www.gurobi.com

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[19] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009, unpublished.

[20] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018.

[21] J. Lee-Thorp, J. Ainslie, I. Eckstein, and S. O. nón, "Fnet: Mixing tokens with fourier transforms," *CoRR*, vol. abs/2105.03824, 2021.