

Joint Optimization of Algebraic Transformation, Device Placement, and Network Bandwidth Allocation for Distributed DNN Training

by

© Hong Wang

B.Sc., Memorial University of Newfoundland, 2023

Supervisor: Kaiyang Liu

Co-supervisor: Qiang Ye

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Master of Science

Department of Computer Science
Memorial University of Newfoundland

October 2025

Abstract

Deep neural networks (DNNs) have been widely adopted across a broad range of application domains, including autonomous driving and recommendation systems. To achieve improved predictive performance, the sizes of training datasets and model parameters have increased substantially. As the sizes of DNN learning models increase, training on a single device is impractical due to limitations in memory and computational resources. To address this challenge, training must be scaled across multiple devices within a distributed cluster. Therefore, distributed training has been adopted to enable efficient training of large DNN models in such environments.

Model parallelism is a widely used strategy for distributed DNN training that partitions model parameters across devices. However, the traditional model parallelism approach suffers from low hardware utilization because only one device can work at a time. To improve utilization, tensor, pipeline, and inter-operator parallelism strategies have been proposed. Among these approaches, inter-operator parallelism necessitates the optimization of device placement and scheduling to minimize training latency. However, determining an optimal device placement is an NP-hard problem. As the DNN model sizes increase, the search space for placement expands exponentially, resulting in significant placement search latency.

In addition, the communication traffic pattern of a DNN training job is determined by the parallelization strategy, leading to non-uniform traffic distributions across inter-server communication links. However, traditional Clos-based topologies, which provide uniform bandwidth and latency, become suboptimal under these conditions. Specifically, links connecting servers that frequently synchronize may become congested, while other links remain underutilized. This imbalance leads to poor network resource utilization and can significantly hinder training performance.

Considering these challenges discussed above, two problems are addressed in this thesis: 1) with non-uniform traffic distribution in DNN training jobs, how to improve the network resource utilization to minimize the communication costs, and 2) with the NP-hard nature of device placement search problem and increased model sizes, how to

determine an efficient device placement decision in a polynomial time. First, to improve network resource utilization, we present joint optimization of device placement and network bandwidth allocation for the first time to accelerate large-scale distributed DNN training. We propose a novel approach to implement the network bandwidth allocation. On each network interface, Open vSwitch (OVS) enforces a Quality of Service (QoS) policy and corresponding flow rules to route egress traffic and allocate bandwidth for each destination server. Simultaneously, we leverage policy-based routing with custom routing tables to simultaneously utilize available bandwidth across multiple physical links if bandwidth demand from a source server exceeds the capacity of a single link to the destination server. Second, to tackle the NP-hardness of the joint optimization problem, we propose a novel algebraic transformation framework based on iterative operator fusion and co-location. This framework generates a compact representation of the DNN computation graph, significantly reducing the search space for device placement and network bandwidth allocation optimization, while incurring minimal performance degradation in training latency.

We evaluate our approach using real-world DNN benchmarks, demonstrating up to a 22% reduction in training latency. Incorporating network bandwidth allocation can provide up to an additional 11% reduction in training latency. More importantly, our design achieves up to 650 times lower solution search latency compared with state-of-the-art methods.

Acknowledgements

Upon the completion of my Master's program, I have many people to thank. First, I would like to express my heartfelt gratitude to my supervisor, Prof. Kaiyang Liu, and to my co-supervisor, Prof. Qiang Ye, for guiding me into academic research. Under their supervisions, I learned how to conduct research with rigor, curiosity, and perseverance.

I would also like to extend my sincere thanks to my trusted labmate whose valuable academic suggestions and ideas contributed to and motivated my research work. In addition, his cooking skills improve the quality of my life in Newfoundland.

My deepest appreciation goes to my parents, Maofeng He and Jinhai Wang, for their unwavering support and encouragement throughout this journey regardless of distance and time. I also appreciate the financial support.

I will continue to work diligently and maintain my passion for research in my next Ph.D. journey.

Contents

Abstract	ii
Acknowledgements	iv
List of Tables	vii
List of Figures	viii
1 Introduction	1
1.1 Background of Distributed DNN Training	2
1.2 Algebraic Transformation	4
1.3 Network Bandwidth Allocation	5
1.4 Thesis Outline	7
2 Related Work	9
2.1 Data and Model Parallelism	9
2.2 Device Placement	11
2.3 Joint Optimization	12
3 System Model and Problem Formulation	15
3.1 Distributed DNN Training	15
3.2 Clos-based Distributed DNN Training Cluster	17
3.3 Placement Optimization Problem Formulation	18
3.4 Clusters with Asymmetric Traffic Distribution	21
3.5 Co-optimization Problem Formulation	25

4	Solution Design	26
4.1	Parallelizable Pairs and Operator Fusion	26
4.2	Operator Co-location	32
4.3	Iterative Fusion and Colocation	33
5	Performance Evaluation	38
5.1	Experimental Setup	38
5.2	Implementation	39
5.3	Baseline	39
5.4	Result Analysis	41
6	Conclusions and Future Works	46
	Appendix A	48
	A Selected Publications	48
	Bibliography	49

List of Tables

3.1	All notations in optimization problems	19
5.1	Average placement search latency comparison in seconds	43
5.2	Comparison of placement search latency under different search space reduction schemes in seconds	45

List of Figures

1.1	Column-wise tensor parallelism	4
1.2	Fat-tree topology	5
3.1	Device placement and scheduling in inter-operator parallelism	16
3.2	Network bandwidth allocation scheme	23
4.1	System model	27
4.2	Scenarios of operator fusion	29
4.3	CDF of operator computing latency	30
4.4	Fusion of operators with mitigated contributions for training latency reduction	31
4.5	Fusion threshold and performance degradation	31
4.6	Challenge of operator fusion with WCC	33
4.7	Iterative operator fusion and co-location	34
4.8	Weakly connected component expansion	35
5.1	Performance of network bandwidth allocation	41
5.2	Benefits of network bandwidth allocation across different link capacities	42
5.3	Per-iteration training latency comparison	44
5.4	Performance loss comparison among search space reduction schemes	45

Chapter 1

Introduction

Since deep neural networks (DNNs) were first introduced, they have become widely used in many areas such as autonomous driving, computer vision, natural language processing, and recommendation systems. The rise of deep learning has led to the development of various DNN structures. The evolution of neural networks began with Feedforward Neural Networks, which process fixed-size inputs without understanding order or structure. To address spatial patterns in data like images, Convolutional Neural Networks (CNNs) [1] were introduced, using filters to capture local features. However, CNNs are not suitable for sequential data, prompting the development of Recurrent Neural Networks (RNNs) that maintain a memory of previous inputs. However, RNNs struggle with long-term dependencies due to exploding or vanishing gradients. Long Short-Term Memory (LSTM) [2] and Gated Recurrent Units (GRU) [3] improve upon RNNs by using gating mechanisms to better capture long-term dependencies. Despite their memory capabilities, LSTM and GRU are inherently sequential, limiting training efficiency. Transformer [4] solves this by using self-attention mechanisms to model global dependencies in parallel, enabling faster and more powerful sequence modeling.

These models have delivered state-of-the-art (SOTA) results in many specialized tasks. For example, CoCa [5], proposed in 2022, achieves a SOTA top-1 accuracy of 91.0 % on ImageNet with a fine-tuned encoder. In general, deep learning has made rapid progress and is being applied to an increasingly wide range of complex application scenarios. This

success often results from using larger training datasets and deeper models to improve prediction performance.

1.1 Background of Distributed DNN Training

To achieve better predictive performance, training datasets and DNN models have increased dramatically in size over the past few years. For example, VGG16 [6] proposed in 2014 has more than 138 million parameters, and Llama-3 [7] 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. However, training such large-scale models on a single device such as a GPU, CPU, or tensor processing unit is impractical. The computational resources of a device refer to its memory capacity and its computational throughput. Memory capacity constrains the number of parameters and intermediate activations that can be stored during training, while computational throughput, typically quantified in floating-point operations per second, determines the rate at which numerical operations can be executed. In practice, the memory available on a single device is insufficient to accommodate all model parameters, and its computational throughput is inadequate for timely training. As a result, training can take days, weeks, or even months to complete. For example, Lambda Labs estimated a hypothetical cost of around \$4.6 million in US dollars and 355 years to train GPT-3 on a single GPU in 2020 [8]. To enable efficient training of large DNN models, distributed training has been widely adopted to accelerate training on scale across a growing number of devices.¹

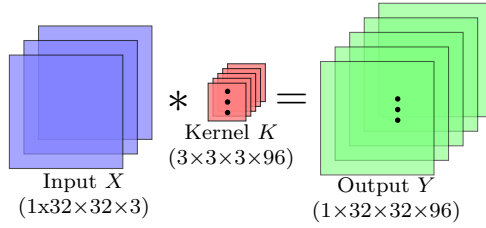
Two commonly used methods for distributed training are data parallelism and model parallelism. In data parallelism, the training dataset is partitioned among devices, each of which maintains a full copy of the model. Although data parallelism enables the scaling of DNN training workloads, pure data parallelism is becoming suboptimal for large training jobs because the cost of synchronizing model parameters among devices will in-

¹Throughout the remainder of this thesis, the terms *device* and *GPU* are used interchangeably, since we focus on clusters consisting of GPUs.

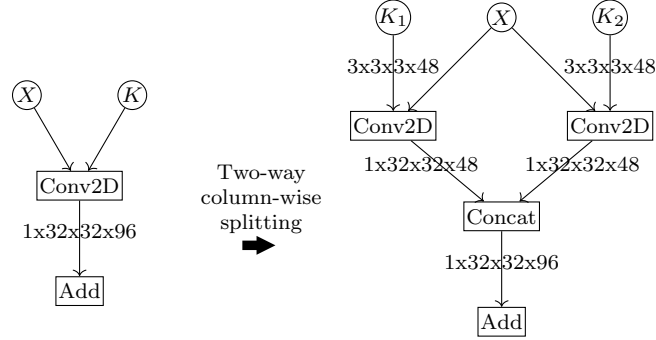
crease significantly as the number of workers grows, resulting in diminishing scalability. More importantly, as models continue to grow in size, the memory capacity of a single device becomes insufficient to accommodate the entire model. To address these challenges, model parallelism partitions the model itself across multiple devices, allowing each device to store and compute only a portion of the model. Compared to data parallelism, traditional model parallelism [9], proposed in 2012, significantly reduces the amount of data transmitted among GPUs. However, it leads to a low hardware utilization rate because only one device can work at a time.

To improve device utilization rate, recent model parallelism strategies are proposed and can be further classified into pipeline parallelism [10, 11, 12], column-wise tensor parallelism [13, 14], as shown in Fig 1.1, and inter-operator parallelism [15, 16, 17, 18]. An operator is a fundamental computational unit in DNNs. It performs mathematical operations such as matrix addition, multiplication, or convolution, and collectively these operators form the computation graph of a model. Pipeline parallelism performs model partitioning at the layer level. However, due to the sequential dependencies between DNN layers, it fails to exploit parallelization opportunities between operators without inter-dependencies. This limitation leads to an increase in the idle time of the device. Meanwhile, tensor parallelism, which partitions individual operators across devices, introduces significant communication overhead due to the need for data exchange at both the split and merge of the operator. In contrast, inter-operator parallelism exploits fine-grained parallelism by distributing independent operators across devices with lower communication overhead than tensor parallelism. This thesis focuses on inter-operator parallelism, which requires the optimization of device placement, specifically the mapping of submodels to GPUs, to minimize training latency.

However, modern DNN models often consist of tens of thousands of operators [16, 19, 20], resulting in an enormous search space for inter-operator parallelism when determining optimal device placement strategies. A DNN model with n operations and a cluster of m GPUs leads to m^n possible placement strategies. The search space grows exponentially with the operators and devices, rendering exhaustive search computationally infeasible.



(a) Example of convolution operation



(b) Computation graph of convolution before column-wise splitting

(c) Computation graph of convolution after column-wise splitting

Figure 1.1: This figure illustrates the application of column-wise tensor parallelism to a Conv2D operation. Consider an input feature map denoted by X with dimensions $1 \times 32 \times 32 \times 3$, and a kernel filter of size $3 \times 3 \times 3 \times 96$. A column-wise tensor parallelism partitions the kernel into two equal halves, enabling parallel computation.

1.2 Algebraic Transformation

Determining the optimal device placement strategy is an NP-hard problem [13]. To mitigate this challenge, algebraic transformations can be applied to reduce the number of operators, thereby enhancing efficiency as the complexity of identifying parallelism increases with the number of operators. As another dimension for the optimization of distributed DNN training, algebraic transformation refers to the optimization of mathematical operations within a neural network's computation graph. The most fundamental form of this optimization is operator fusion, which eliminates the need to store intermediate results in memory and reduces the frequency of memory accesses [21]. Taking convolution and bias addition as an example, fusion allows the bias addition to be integrated directly into the convolution kernel. This reduces redundant memory accesses and improves execution efficiency. Moreover, operator fusion reduces the number of decision

variables when identifying optimal device placement for inter-operator parallelism. To mitigate the scalability challenge introduced by device placement and network bandwidth optimization, we apply algebraic transformations to generate a compact representation of the DNN computation graph with reduced search space.

1.3 Network Bandwidth Allocation

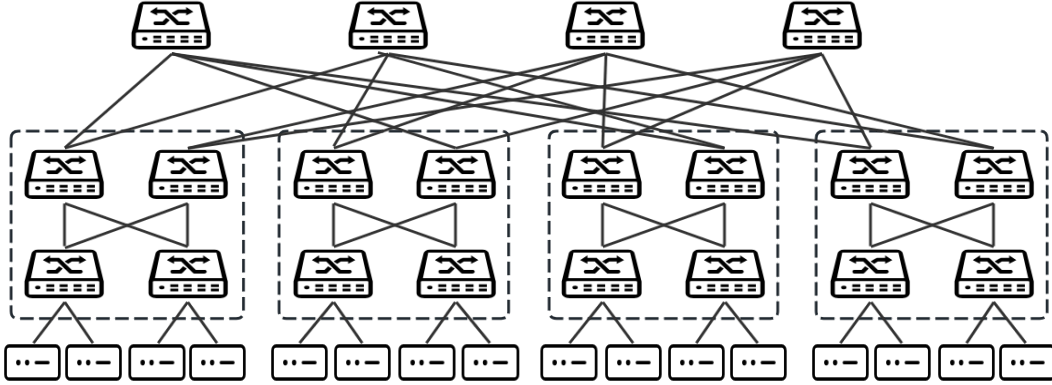


Figure 1.2: An example of Fat-tree topology [22] that provides uniform bandwidth among each server pair.

Optimizing device placement decisions leverages parallelization opportunities among operators, enabling concurrent execution to minimize training latency. However, scaling distributed training clusters introduces significant challenges because the communication overhead of large DNN training jobs increases dramatically with the number of workers [23]. For instance, in Meta’s production environment, communication can account for up to 60% of the time per training iteration [23]. The communication traffic pattern of a DNN training job is determined by the parallelization strategy [23], leading to non-uniform traffic distributions across inter-server communication links.

Traditional symmetric spine-leaf data center network architectures, such as Fat-Tree [22], as shown in Fig. 1.2, are designed to provide uniform bandwidth and latency between server pairs. Thus, they are suboptimal for supporting the intensive synchronization traffic and inherently non-uniform communication patterns characteristic of DNN workloads,

resulting in inefficient utilization of network resources. To address this limitation, this work proposes a bandwidth allocation scheme that accommodates non-uniform traffic patterns by allocating bandwidth resources asymmetrically across inter-server communication pairs. Specifically, more bandwidth resource is reserved for communication from a source server that demands higher egress traffic toward a destination server. To achieve this, Open vSwitch (OVS) [24, 25] is implemented at the server side to implement a Software-Defined Networking (SDN) architecture [26]. OVS enables efficient source-based and applies a Quality of Service (QoS) policy on each interface for bandwidth allocation. When the bandwidth demand from a source server exceeds the capacity of a single link to the destination server, policy-based routing directs flows from each source IP to their designated network interfaces, enabling simultaneous utilization of multiple physical links. This approach enables the aggregate bandwidth between a source server and a destination server to exceed the maximum achievable over a single link in traditional Clos-based networks. It also alleviates contention when multiple flows are transmitted simultaneously on the same link, further reducing communication costs.

Motivated by these observations, the co-optimization of algebraic transformation, device placement, and network bandwidth allocation could achieve optimal performance in training latency. To address the computational complexity arising from large search space, we propose an iterative operator fusion and co-location scheme to reduce the search space while incurring minimal performance degradation in training latency, leading to an efficient joint optimization of algebraic transformation, device placement, and network bandwidth allocation for large-scale distributed DNN training. Performance evaluation on four real-world DNN benchmarks demonstrates that the joint optimization of algebraic transformation and device placement can reduce per-iteration training latency by up to 22%. Incorporating additional network bandwidth allocation can provide up to an additional 11% reduction in training latency. More importantly, our design achieves up to 650 times lower solution search latency compared with SOTA methods.

In summary, this thesis makes the following contributions.

- This thesis is the first work that jointly optimizes device placement and network

bandwidth allocation to accelerate distributed DNN training.

- To tackle the challenges of NP-hardness and ever-increasing model sizes, this thesis proposes a novel algebraic transformation framework based on iterative operator fusion and co-location. This approach significantly reduces the search space generated by device placement and network bandwidth allocation optimization, while incurring minimal performance degradation in training latency.
- We propose a novel network bandwidth allocation scheme to accommodate the non-uniform traffic distribution in distributed DNN training. This approach enforces QoS policies through OVS configurations and policy-based routing with custom routing tables on each network interface to achieve the desired bandwidth allocation. This scheme overcomes the limitations of traditional Clos-based topologies (e.g., spine-leaf), which provide uniform bandwidth and latency but are constrained by single-link bandwidth ceilings. Network bandwidth allocation can be extended to servers interconnected through RDMA over Ethernet (RoCE) using link aggregation.

1.4 Thesis Outline

Chapter 1 contains a distributed DNN training background, research topics, and contributions to this thesis, followed by the structure of the thesis.

Chapter 2 presents a comprehensive review of related works, encompassing topics from data parallelism to model parallelism. In particular, it categorizes existing approaches to operator parallelism and provides a detailed summary of the advantages and limitations associated with each category.

Chapter 3 addresses the challenge of inefficient resource utilization in Clos-based network architectures. We consider a computing cluster similar to the one studied in TopoOpt [23], where there are multiple disjoint paths between each pair of servers. Within this topology, we propose a network bandwidth allocation scheme designed to fully leverage these redundant paths and accommodate the non-uniform traffic distribu-

tion characteristic of DNN training jobs. Then, a mixed integer programming (MIP) problem is formulated to jointly optimize the device placement decision and network bandwidth allocation.

Chapter 4 addresses the challenge of NP-hardness of the joint optimization of device placement decision and network bandwidth allocation. A novel algebraic transformation framework based on iterative operator fusion and co-location is proposed. This approach significantly reduces the search space generated by device placement and network bandwidth allocation optimization, while incurring minimal performance degradation in training latency. Then, a new IP problem with a significantly reduced search space is reformulated.

Chapter 5 presents experiments on network bandwidth allocation and simulation for parallelization, and then combines these components for comprehensive emulation. It first validates the effectiveness of the proposed network bandwidth allocation scheme and then quantifies the per-iteration training latency and solution search latency across different GPU counts. Finally, it evaluates the benefits of network bandwidth allocation under diverse link bandwidth configurations, demonstrating how the proposed approach improves performance in varying network environments.

Chapter 6 concludes the thesis and discusses the future work.

Chapter 2

Related Work

2.1 Data and Model Parallelism

Two fundamental approaches for distributed DNN training are data and model parallelism. For data parallelism [27, 28], each worker maintains a full copy of the model but trains on disjoint subsets of the training data, computing local gradients based on its local data subset. The communication pattern among different devices in data parallelism can be classified as parameter servers (PS) and AllReduce. In PS, where one master node and multiple slave nodes form a centralized structure, the master node will accumulate the local gradients computed on all workers and then update the global weights. Then, the latest global weights are sent to all workers for the next iteration. On the other hand, AllReduce, strictly synchronous, is a decentralized structure that enables inter-workers communications. The Ring AllReduce [29] algorithm, originally proposed by Baidu, completes an AllReduce operation in two phases including *Reduce-Scatter* and *All-Gather*. Each phase requires exactly $p - 1$ communication rounds, where p denotes the number of participating devices. During each round, a device communicates only with its two immediate neighbors in a logical ring topology, ensuring high bandwidth efficiency. In contrast, the recently proposed Tree AllReduce algorithm achieves even greater communication efficiency. It completes the entire AllReduce operation in just $2 \log_2 p$ steps by organizing devices in a binary tree structure and performing a reduction phase followed

by a broadcast phase. This logarithmic communication complexity makes Tree AllReduce particularly advantageous in large-scale distributed systems. However, in both PS and AllReduce structures, every node typically needs to send the entire model’s gradients in each iteration. Thus, pure data parallelism is suboptimal for training modern large-scale deep neural networks because it faces limitations in memory capacity on individual devices and experiences significantly increased communication overhead as the number of workers grows. These issues lead to poor scalability and reduced training efficiency.

Model parallelism [9] was introduced in 2012 to address the challenge of DNN models that are too large to fit within the memory of a single device. By partitioning the model layers across multiple workers, the memory burden is distributed, and the overall communication volume is reduced. However, in each iteration, each GPU must wait for the preceding GPU to complete its forward and backward passes on a mini-batch before beginning its computation, leading to wasted GPU resources since only one GPU is active at a time. To improve the GPU utilization, pipeline parallelism [10, 11, 12] enables mini-batch or even micro-batches to be executed in a pipeline manner. A GPU will process a subsequent mini-batch after completing a previous one. Thus, multiple GPUs can work at the same time to increase the training throughput. However, this process may introduce staleness in the asynchronous case where gradients are computed using outdated model parameters. Also, due to the sequential dependencies between DNN layers, it fails to exploit parallelization opportunities between operators without inter-dependency. This limitation leads to suboptimal resource utilization and increased device idle time. Inter-operator parallelism [15] is then designed to further parallelize these parallelizable operators, leading to performance improvements by reducing idle times on hardware resources. In inter-operator parallelism, optimizing device placement decisions and operator scheduling minimizes training latency [16].

2.2 Device Placement

Device placement [19], originally proposed in 2017, plays a critical role in distributed DNN training by determining how computational operations are assigned to physical devices. Existing approaches can be broadly categorized into heuristic, learning-based, and solver-based methods. Heuristic schemes, while computationally efficient, suffer from a high performance loss in preserving low training latency, while learning- and solver-based schemes face high computational complexity.

Heuristic solutions: Existing graph partitioning tools, such as Metis [30, 17, 31], divide computation graphs into load-balanced subgraphs for placement while minimizing cross-device data transmission. Additionally, starting from a random solution, local search [32] greedily improves the best single-operator placement reassignment repeatedly until a local optimum is reached. In contrast, Markov Chain Monte Carlo (MCMC) [23, 13] 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 typically suffer from significant performance loss, lacking a worst-case performance guarantee.

Learning-based solutions: Learning-based strategies have been proposed to optimize device placement in computational graphs. Early approaches such as ColocRL[19], HDP [33], and SpotLight [34] adopt reinforcement learning to train RNN/LSTM controllers for placement decisions but these controllers must be retrained for each new computation graph due to their limited ability to capture long-term dependencies. To address this limitation and enable more generalizable placement policies, recent methods such as GDP [31], GO [35], PlaceTo [18], Regal [17], HeteroG [36] and TAG [20] use GNN to extract and learn from structural information of computation graphs. However, all these learning-based methods require extensive training time for the model to produce placement decisions that outperform existing heuristics, particularly in RNN/LSTM-based approaches, which generate per-operator placements sequentially.

Solver-based solutions: Solver-based approaches generally face high computational complexity and search latency when addressing large-scale, NP-hard device placement

problems. For example, Pesto [16] formulates device placement and scheduling as an integer linear programming problem and employs batch merging to accelerate the solution search process. Nevertheless, the joint optimization of placement and scheduling remains computationally intensive due to the NP-hard nature of both problems. Moreover, aggressive batch merging diminishes opportunities for parallel execution, leading to performance degradation in training latency. Tarnawski *et al.* [37] also framed placement as an IP problem and improve parallelism through the use of non-contiguous partitioning. However, their method is primarily effective for small-scale scenarios and does not account for heterogeneous network environments. To achieve an effective device placement decision within a polynomial time, our design exploits the parallelization opportunities among parallelizable operator pairs and applies a novel algebraic transformation to reduce the search space while minimizing performance loss in training latency.

2.3 Joint Optimization

Device Placement \times Network Topology: As a pioneering research work, TopoOpt [23] alternately optimizes the parallelization strategy and the topology between servers offline until convergence is achieved. Subsequently, it reconfigures the optical switches to implement the target topology. By finding multiple permutations for each AllReduce group and overlapping their corresponding sub-topologies, TopoOpt efficiently completes large transfers for data parallelism traffic and lowers the hop count for model parallelism transfers. However, the iteration-by-iteration alternating optimization of parallelization and topology may lead to convergence at a local optimum. Additionally, exploring either search space, using MCMC, is computationally intensive and time-consuming. The number of iterations until convergence increases as the search space expands.

Device Placement \times Algebraic Transformation: To reduce the search space with tens of thousands of operators in a computation graph, graph coarsening schemes are employed to lower placement search latency. However, previous schemes often aggressively reduce the number of operators without considering parallelization opportunities.

ColocRL [19] assigns two operators to the same device if the output of an operator is exclusively consumed by its immediate successor. PlaceTo [18] merges the operator with the lowest cost into one of its successors. If the operator has no successors, it is instead merged into its predecessor. SpotLight [34] groups operations that share the same two-level prefix. In contrast, Pesto [16] employs batch merging to merge thousands of vertices in a single batch. Although these approaches effectively reduce search space, they also result in a significant decrease in performance to preserve low training latency due to the loss of parallelization opportunities.

Existing works primarily focus on device placement optimization to accelerate distributed DNN training, but they overlook the low network resource utilization caused by the uniform traffic distribution of DNN workloads. To address this limitation, TopoOpt [23] is the first work to co-optimize parallelization strategies and the underlying physical network topology. However, its practicality is limited by the reliance on expensive configurable optical switches. Also, the optimization procedure in TopoOpt is inherently sequential. In each iteration, only one dimension is optimized at a time, and all dimensions must be traversed sequentially. This process requires numerous iterations to achieve convergence. As the search space grows, the overall time and computational resources required increase significantly. We focus on the same topology studied in TopoOpt. To enhance practicality, we seek to eliminate the need for expensive hardware, such as optical switches. Furthermore, to overcome the slow convergence inherent in sequential iterative processes, we propose a joint optimization of device placement and network bandwidth allocation decisions. However, achieving effective co-optimization is particularly challenging because optimizing device placement decisions is an NP-hard problem. Prior methods often rely on algebraic transformation schemes to reduce the search space. However, existing approaches often fail to preserve parallelization opportunities, as aggressive transformations tend to cluster parallelizable operators onto a single device, while their sequential execution leads to underutilization of computational resources. To mitigate the resulting performance loss in training latency, we propose an iterative fusion and co-location scheme that progressively merges operators with limited or no loss

of inter-parallelization opportunities and co-locates those contributing the highest cumulative computation and communication costs along the critical path. This strategy substantially reduces the search space generated by the co-optimization of device placement and network bandwidth allocation decisions, while incurring minimal performance degradation in training latency.

Chapter 3

System Model and Problem

Formulation

3.1 Distributed DNN Training

In inter-operator parallelism, a DNN computation graph is partitioned into disjoint sub-graphs. To minimize overall training latency, it is crucial to explore efficient strategies for both device placement, which maps each subgraph to a specific device, and operator scheduling, which determines the execution order of operators within each subgraph. An effective placement parallelizes operators without data dependency, enabling more tasks to be processed simultaneously. Similarly, efficient scheduling reduces device idle time by minimizing delays in waiting for data from other devices.

The DNN computation graph is represented as a directed acyclic graph (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, with a transmission tensor size of $t_{i,j}$. For a pair of operators with data dependency, such as A and B in Fig. 3.1(a), operator B depends on operator A and cannot be executed until the output of operator A is available as its input. Fig. 3.1(a) also uses dashed ellipses to indicate parallelizable operator pairs with no directed path connecting them. For a pair of parallelizable operators, such as B

and C, one operator can be executed before or after the other, generating two feasible topological sorts (see Fig. 3.1(b)).

A topological sort yields a valid operator scheduling by producing a linear ordering of operators in which each operator precedes all operators that depend on its output. The existence of a topological sort is both a necessary and sufficient condition for the graph to be a DAG. Fig. 3.1(c) shows all placement schemes on two GPUs. Assume that the computing delays of operator A, B and C are $5 \mu\text{s}$, $10 \mu\text{s}$, and $5 \mu\text{s}$, respectively. In addition, the cost of communication between devices is assumed to be $5 \mu\text{s}$. The middle configuration in Fig. 3.1(c) illustrates the optimal placement where the nodes in the parallelizable operator pair, B and C, are on different GPUs to enable parallel execution, leading to the shortest per-iteration training latency of $15 \mu\text{s}$. This example demonstrates that device placement significantly impacts distributed training performance, which can often be formulated as an IP problem with NP-hard properties [16]. The search space for an optimal device placement expands rapidly with the scale of DNN models and the number of devices.

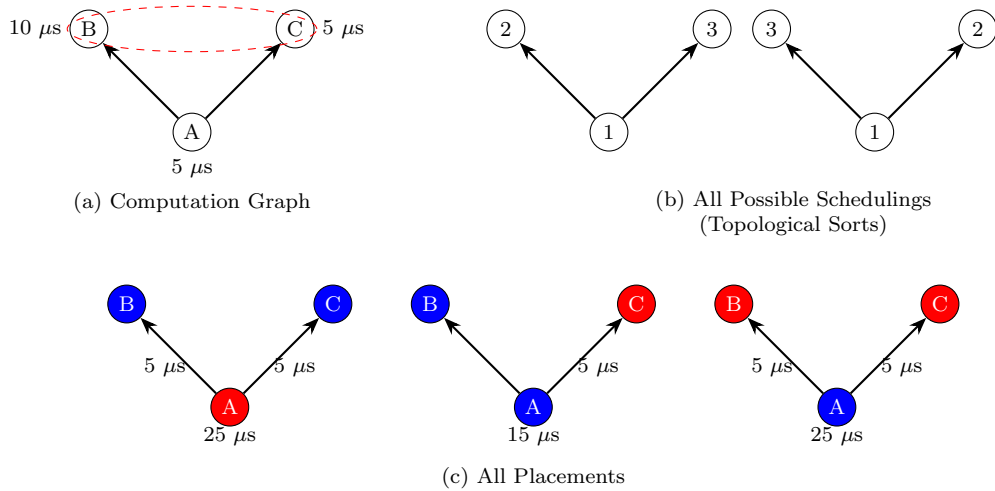


Figure 3.1: The dashed ellipse in (a) represents a pair of parallelizable operators. Multiple topological sorts in (b) are possible due to these parallelizable operator pairs. In (c), red and blue nodes indicate tasks running on GPU 1 and 2, respectively. We assume each cross-GPU communication takes five microseconds. The number below each placement represents the training latency under the optimal scheduling.

Communication cost model: Since inter-operator parallelism does not involve collective communication patterns such as `AllReduce`, we consider only peer-to-peer communication. Following the approach used in Pesto [16], FlexFlow [13], and Astra-sim [38], we model the communication cost using the standard α - β cost model, expressed as $p_0 + \frac{t_{i,j}}{b_{d,e}}$, where p_0 represents the fixed connection setup latency, and the fraction denotes the data transmission time between operators i and j , based on the data size $t_{i,j}$ and the bandwidth $b_{d,e}$ of the communication link from device d to e .

Placement decision and bandwidth allocation: Based on the input computation graph G and device set D , we define a placement strategy \mathcal{P} as $\{(g_d, d) \mid d \in D, g_d \in \mathcal{G}\}$ 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. When considering the joint optimization of device placement decisions and network bandwidth allocation, an asymmetric bandwidth allocation strategy is formally defined as a set $\mathcal{B} = \{(d, e, b_{d,e}) \mid d, e \in D\}$, where $b_{d,e}$ denotes the egress bandwidth allocated from device d to device e for each device pair (d, e) .

3.2 Clos-based Distributed DNN Training Cluster

This thesis considers a homogeneous computing system where all servers are equipped with identical GPUs. Such configurations are commonly used in large-scale training environments, including those employed by Meta [7] and NVIDIA [14]. The uniformity in hardware simplifies system architecture, and facilitates efficient scaling. Given that different learning tasks can impose distinct resource and security constraints, potentially leading to substantial and unpredictable performance degradation when executed concurrently [39], we assume that each GPU is allocated to a single learning task at any given time.

A directed graph $T = (D, L)$ represents the device topology, where D is the set of devices, and L represents the inter-server communication links. Each server $d \in D$ has a maximum memory capacity M_d . Servers are interconnected via electrical packet switches

arranged in a multi-tier Fat-Tree network topology [22], 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 . 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.

Profiling: To maximize the efficiency of DNN training pipelines, it is essential to balance computationally intensive and I/O-intensive tasks across available hardware resources. Since CPUs are not optimized for computationally intensive operations due to their limited core count, dataset-related operators that are primarily I/O-intensive are typically assigned to the CPU, whereas computationally intensive operators are executed on GPUs. By offloading data preprocessing to the CPU, the system can prepare the next batch in parallel while the GPU executes training tasks, thereby overlapping I/O and computation to maximize overall throughput. Then, profiling is performed on the GPU to estimate the training latency and memory consumption of the remaining operators in the computation graph. This profiling leverages TensorFlow’s built-in tools to ensure seamless integration into the training workflow.

3.3 Placement Optimization Problem Formulation

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

Table 3.1: All notations in optimization problems

Notation	Description
$x_{i,d}$	Variable indicating if operator i is allocated to device d
R	Set of all parallelizable operator pairs
U	Set of operators in a weakly connected component
W	Set of all weakly connected components identified
Q	An arbitrarily large number
O	Set of all operator pairs $\langle i, j \rangle$ such that $tp_i < tp_j$
A	Per-iteration training latency
B	The upper bound for aggregated egress bandwidth of a server
TP	Topological sort of all operators
tp_i	Topological order of operator i in TP
m_i	Memory cost for operator i
s_i, f_i	Start/finish time of execution for operator i
p_i	Computing delay of operator i
$t_{i,j}$	Size of tensor transmission from operator i to j
$b_{i,j}$	Bandwidth from device i to j
$\delta_{d,e}^{i,j}$	Variable indicating if operator i is on device d and operator j is on device e

Scheduling: the SOTA framework such as Pesto [16], which jointly optimizes device placement and scheduling, faces an exponentially growing search space as the model size increases, due to the NP-hard nature of both device placement and scheduling search problems. It takes a solver approximately 51 minutes to determine the optimal placement and scheduling of a 4-layer neural machine translation model, even after the computation graph is coarsened to approximately 200 operators. To mitigate this issue, the utilization of topological sort enables the optimization to focus on device placement search, preventing the exponentially increasing search space. To obtain a topological sort of the computation graph G , we employ a DFS-based implementation of the topological sort technique, as an alternative to the approach proposed in Kahn’s algorithm [40]. The algorithm produces an ordered operator list $TP = (i_1, i_2, \dots, i_n)$, where $\{i_1, i_2, \dots, i_n\} = V$. 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 . $O = \{\langle i, j \rangle \mid tp_i < tp_j, i, j \in V\}$ lists all unique operator pairs $\langle i, j \rangle$ 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.

$$\mathbf{P1:} \quad \text{Min}_{\{x_{i,d}\}} \quad A = \max_{i \in V} f_i$$

s.t.

$$x_{i,d} \in \{0, 1\}, \quad \forall i \in V, \quad \forall d \in D, \quad (3.1)$$

$$M_d \geq \sum_{i \in V} m_i \cdot x_{i,d}, \quad \forall d \in D, \quad (3.2)$$

$$\sum_{d \in D} x_{i,d} = 1, \quad \forall i \in V, \quad (3.3)$$

$$s_i + p_i = f_i, \quad \forall i \in V, \quad (3.4)$$

$$\begin{cases} f_i \leq s_j, & \text{if } t_{i,j} = 0, \quad \forall (i, j) \in E, \\ f_i + (p_0 + \frac{t_{i,j}}{b}) \cdot (1 - \sum_{d \in D} x_{i,d} \cdot x_{j,d}) \leq s_j, & \text{else,} \end{cases} \quad (3.5)$$

$$f_i \leq s_j + Q \cdot (2 - x_{i,d} - x_{j,d}), \quad \forall (i, j) \in O, \quad \forall d \in D, \quad p_i, p_j > 0. \quad (3.6)$$

In **P1**, $x_{i,d}$ is a binary variable that denotes the mapping between operator i and device d where $x_{i,d} = 1$ if operator i is allocated to device d , and $x_{i,d} = 0$ otherwise. Constraint (3.2) ensures the total required memory of assigned operators does not exceed the device memory capacity M_d of device d . Constraint (3.3) ensures each operator is allocated to one device, forming a many-to-one mapping between operators and devices. Constraint (3.4) 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 (3.5) maintains the global data dependency, ensuring that if there is an edge $(i, j) \in E$, operator j starts execution only after operator i is completed and the corresponding tensor transmission is received. This transmission may incur communication costs, which are calculated based on three possible scenarios:

- 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., $d \neq e$), a corresponding communication cost is incurred. The cost is equal to the connection-setup latency p_0 plus the transmission time, calculated as the tensor size $t_{i,j}$ divided by the bandwidth $b_{d,e}$ from devices d to e . However, if an edge (i, j) carries a tensor of `dtype=resource`, a handle to a TensorFlow resource, then operators i and j must be placed on the same device. Such an edge carries a handle to a device-local state object, ensuring that stateful reads and writes target the same resource instance.

Constraint (3.6) ensures that for each unique operator pair $\langle i, j \rangle \in O$, the operator j can only be executed 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 an arbitrarily large constant. This constraint guarantees non-overlapping computing periods and valid scheduling, complying with the topological sort, on each device.

3.4 Clusters with Asymmetric Traffic Distribution

Traditional Clos-based topologies, such as the spine–leaf architecture, are designed to provide uniform bandwidth and latency across all server pairs. However, these designs are often suboptimal for DNN training workloads, which typically exhibit highly non-uniform, communication-intensive traffic patterns.

To better accommodate the demands of such workloads, we consider a computing cluster similar to the one studied in TopoOpt [23]. As illustrated in Fig. 3.2(a), the cluster consists of multiple servers connected through a layer comprising d switches. Each server is equipped with d network interfaces, each of which connects to a distinct switch. All servers are interconnected using TCP or RoCE, with each link offering a bandwidth of b . For each pair of servers, there are d disjoint paths of equal cost available. Thus, the upper bound of the aggregate egress bandwidth of each server is equal to the sum of bandwidth of all connected physical links, denoted as $B = bd$. By leveraging the multiple disjoint paths enabled by this topology, the aggregate inter-server communication bandwidth can exceed the maximum bandwidth achievable via a single link in traditional Clos-based

networks.

In scenarios where the number of physical network interfaces per server is limited, effective connectivity can be enhanced through the use of NICs that support break-out cables. Break-out cables allow a single high-bandwidth interface (e.g., 100 Gbps) to be split into multiple lower-bandwidth links (e.g., 4×25 Gbps), effectively increasing the number of available connections per NIC. To further scale the system, we introduce a hierarchical interconnect architecture in which servers and their associated switch layer are grouped into pods. These pods are then interconnected via a core network layer organized in a Fat-Tree topology. This forms a direct-connect topology at the ToR or spine layers, consistent with previously proposed architectures.

Similar to the device topology defined in the Fat-tree architecture, we model the computing cluster as a fully connected directed graph. $T = (D, L)$, where D is the set of devices, and L represents the inter-server communication links. Each server $d \in D$ has a maximum memory capacity M_d . However, since network bandwidth allocation is co-optimized with device placement decisions, the egress bandwidth allocated from server d to server e is denoted by $b_{d,e}$.

As the system scales, OVS [25, 24] is deployed on each server to implement SDN, enabling end servers to forward data according to user-defined policies while ensuring non-blocking traffic. By enforcing QoS policies with flow rules on each interface, OVS enables finer-grained traffic distribution and better compliance with desired traffic patterns.

To accommodate the non-uniform traffic patterns inherent in DNN training workloads, we implement asymmetric traffic distribution across disjoint network paths. Specifically, greater bandwidth is reserved for communication from a source server that exhibits higher egress demand toward a destination server. The following network bandwidth allocation scheme is employed to support this design.

- **Qos and traffic shaping:** To enforce predictable bandwidth allocation and mitigate resource contention, egress QoS policies are configured on network interfaces using the Hierarchical Token Bucket (HTB) queuing discipline. Each interface is assigned a QoS policy with a parameter *max_rate* equal to the physical link band-

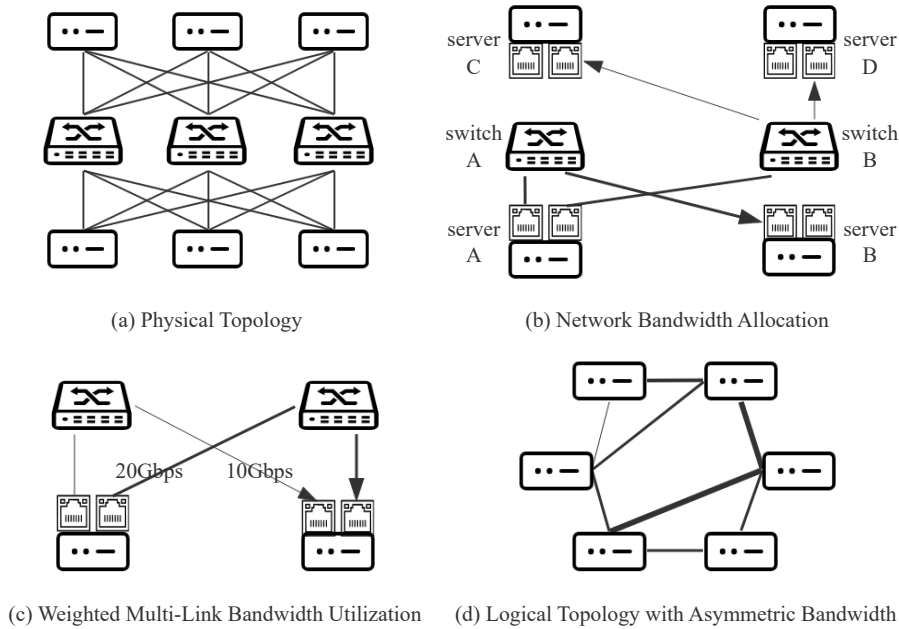


Figure 3.2: (a) is an example of a physical topology consisting of six servers and three switches, where each link has a bandwidth of 20 Gbps, and each server is directly connected to all d switches. (b) is an example where server A communicates with server B at 20 Gbps while simultaneously communicating with servers C and D at 10 Gbps each. (c) is an example where the source server employs concurrent transmission queues across dual network interfaces to forward traffic to the destination server. (d) illustrates the logical topology corresponding to (a) with uneven allocation of bandwidth resources between each pair of communicating servers. In (b), (c), and (d), the thickness of each line represents the bandwidth allocated from the source to the destination server.

width, which serves as a parent container for subordinate queues. Within this structure, if an interface forwards traffic to a single destination, a single queue is defined under the QoS policy, with a queue-level parameter *max_rate* equal to that of the policy itself. In contrast, if an interface forwards traffic to multiple destinations, separate queues are configured under the QoS policy, each with a destination-specific *max_rate* value. The aggregate of these queue-level *max_rate* values for each interface cannot exceed the value of the *max_rate* defined at the QoS policy level. By configuring multiple queues, contention is mitigated when concurrent flows transmit over a shared link toward distinct destination servers.

To illustrate this principle, consider a toy example in Fig. 3.2(b), where each server involves two physical interfaces and the bandwidth of each links is 20 Gbps. For the left interface on server A, a QoS policy is configured with a maximum transmission rate *max_rate* of 20 Gbps. Under this policy, a single queue is created to handle traffic destined for server B, with its maximum rate also set to 20 Gbps. Similarly, for the right interface, a QoS policy is configured with the same *max_rate* equal to the full link capacity. However, within this policy, two queues are each allocated 10 Gbps to manage traffic directed to server C and server D, respectively.

- **Weighted multi-Link bandwidth utilization:** As shown in Fig. 3.2(c), in scenarios where the bandwidth demand from a source server exceeds the capacity of a single link to the destination server, traffic is distributed across multiple links through their corresponding network interfaces. Each flow is transmitted through an interface associated with a queue assigned to the traffic destined for that specific server. For example, consider inter-server communication implemented using TCP. The NVIDIA Collective Communications Library establishes multiple independent TCP connections. On the sender side, policy-based routing uses custom routing tables to match source IP addresses and ensure that flows from each source IP are directed through their designated network interface. When the destination server has multiple reachable IP addresses, each mapped to a different physical network interface, the kernel's IP routing mechanism can naturally distribute these TCP flows across the available interfaces according to the routing table. This approach utilizes the bandwidth resources of multiple links simultaneously, thus reducing the communication cost and enabling our approach to outperform the tradition Clos-based topology whose maximum bandwidth achievable does not exceed single link.
- **Flow mapping:** Finally, each queue is assigned to a traffic flow to a destination server using the `add-flow` command in OVS, forming a one-to-many mapping from destination servers to queues. The static forwarding rules simulate the behavior of SDN controller.

3.5 Co-optimization Problem Formulation

After the profiler calculates the computing delay p_i and memory consumption m_i for each operator $i \in V$, we formulate the device placement and topology search as an MIP problem in **P2**, where A is the finish time of the last completed operator, that is, $\max_{i \in V} f_i$. By solving **P2**, we aim to find an optimal placement strategy \mathcal{P}^* and bandwidth allocation \mathcal{B}^* to minimize the per-iteration latency A for a given computation graph G and computing cluster T . The mathematical formulation of **P2** is shown below.

$$\mathbf{P2:} \quad \text{Min}_{\{x_{i,d}\}, \{b_{d,e}\}} \quad A = \max_{i \in V} f_i$$

s.t.

Constraints (3.1), (3.2), (3.3), (3.4), (3.6)

$$\sum_{e \in D, d \neq e} b_{d,e} \leq B, \quad \forall d \in D, \quad (3.7)$$

$$\begin{cases} \delta_{d,e}^{i,j} \in \{0, 1\}, \\ \delta_{d,e}^{i,j} \geq x_{i,d} + x_{j,e} - 1, \quad \forall (i, j) \in E, d, e \in D, d \neq e \\ \delta_{d,e}^{i,j} \leq x_{i,d}, \quad \delta_{d,e}^{i,j} \leq x_{j,e}, \end{cases} \quad (3.8)$$

$$\begin{cases} f_i \leq s_j, & \text{if } t_{i,j} = 0, \\ f_i + \sum_{d, e \in D, d \neq e} \delta_{d,e}^{i,j} \cdot (p_0 + \frac{t_{i,j}}{b_{d,e}}) \leq s_j, & \text{else,} \end{cases} \quad \forall (i, j) \in E, \quad (3.9)$$

In **P2**, in addition to the decision variable $x_{i,d}$ defined in **P1** that indicates mapping between operator i and device d , another decision variable $b_{d,e}$ is introduced to represent the egress bandwidth allocated from device d to device e . Constraint 3.7 ensures that for each device $d \in D$, the total outgoing bandwidth $\sum_{d \in D} b_{d,e}$ does not exceed the capacity of all connected physical links. In contrast to constraint 3.5, $b_{d,e}$ is a decision variable instead of a parameter in constraint 3.9, the term $x_{i,d} \cdot x_{j,d} \cdot (p_0 + \frac{t_{i,j}}{b_{d,e}})$ constitutes a cubic programming problem, which is inherently non-convex and significantly more challenging to solve to global optimality compared with quadratic programming problems. Thus, we introduce $\delta_{d,e}^{i,j}$ to mitigate the non-linearity arising from the term $x_{i,d} \cdot x_{j,d} \cdot \delta_{d,e}^{i,j} = 1$ if operator i is placed on device d and operator j is placed on device e .

Chapter 4

Solution Design

Since finding the optimal parallelization strategy alone is an NP-hard problem [13], and considering the allocation of network bandwidth further complicates this problem, finding an optimal solution of **P2** within polynomial time is infeasible. Typically, the solution search latency of commercial solvers increases exponentially with the size of the problem [13]. Therefore, we propose a novel iterative operator fusion and co-location scheme to reduce the search space of **P2** while minimizing the subsequent performance loss in training latency, significantly decreasing solution search latency. Fig. 4.1 shows the workflow of our proposed solution framework. A condensed computation graph with a significantly reduced search space is generated after the proposed iterative operator fusion and co-location scheme. Upon resolution of the MIP problem, which determines the optimal bandwidth allocation policy and device placement decisions, operators are deployed accordingly. Subsequently, QoS policies implemented through OVS configurations and policy-based routing with custom routing tables are applied on each network interface to achieve the desired bandwidth allocation.

4.1 Parallelizable Pairs and Operator Fusion

Definition 1. *In a DAG $G = (V, E)$, an operator pair $\langle i, j \rangle$, 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.*

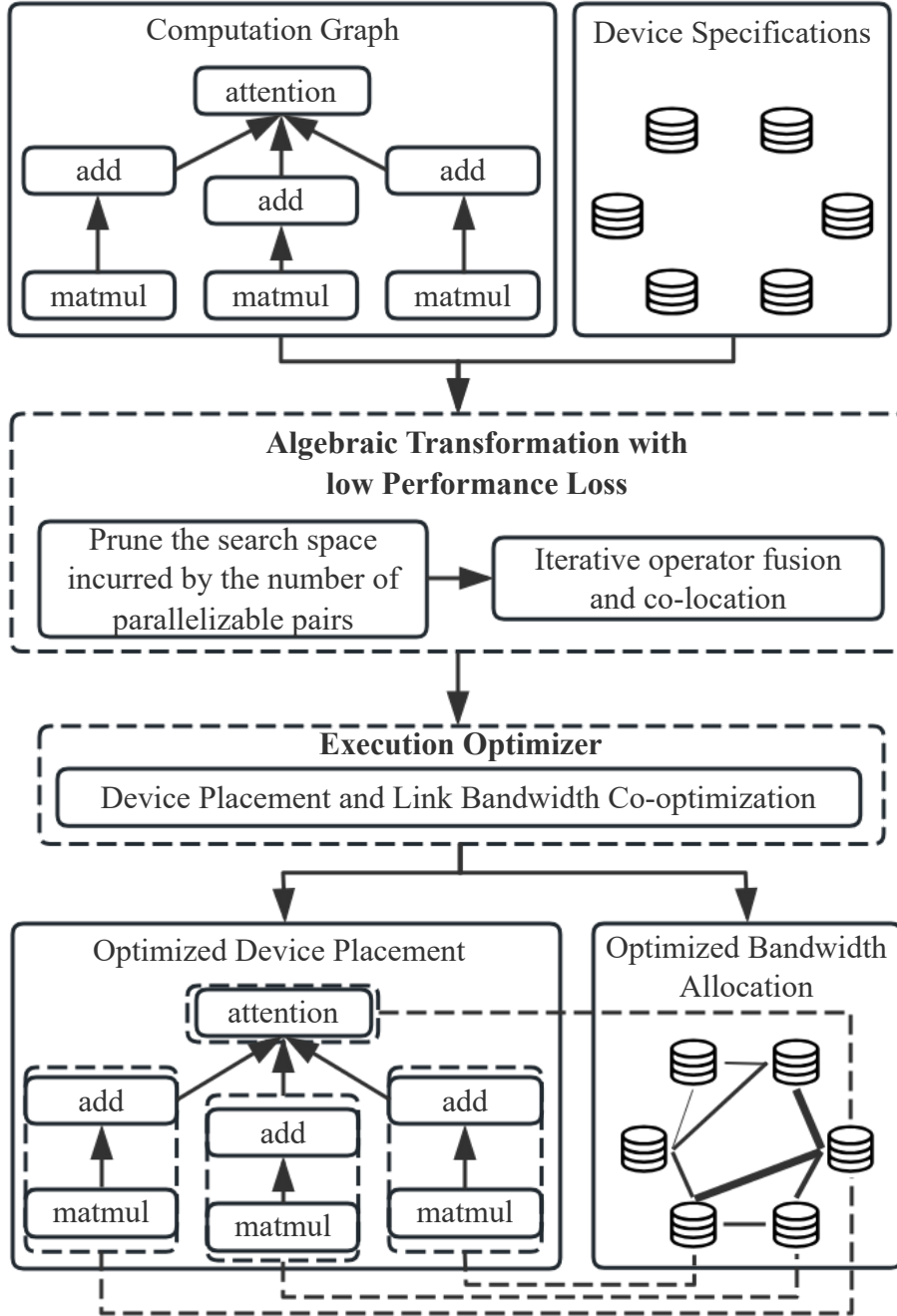


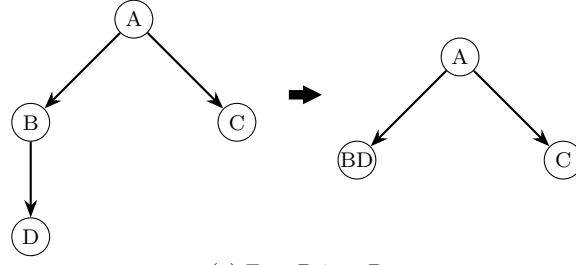
Figure 4.1: Our solution consists of a profiler to evaluate the computing and memory costs of operators. Following the reduction of the search space through operator fusion and co-location schemes, an MIP model explores the pruned search space, generating an efficient strategy for device placement and network bandwidth allocation.

Existing solver-based approaches such as Pesto [16] optimize over all possible operator pairs, which can be computationally expensive. To reduce the complexity of the constraint (3.6), we exclude any operator pair $\langle i, j \rangle$ for which there exists a directed path from i to j in the computation graph. According to the topological sort rule, if such a path exists, operator i must precede operator j in any valid topological ordering of the DAG. Therefore, these pairs are already implicitly ordered by the global data dependency constraint (3.5). This selective consideration significantly reduces the computational complexity of **P2**.

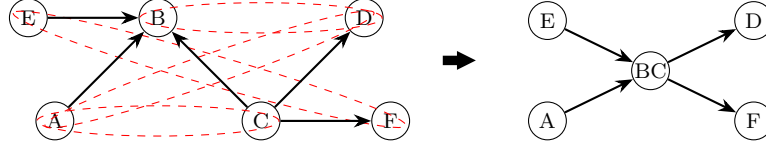
The DNN computing graph often includes thousands or even tens of thousands of operators for large learning models [19, 16]. We aim to fuse operators to reduce their number, accelerating the placement search process. One challenge of operator fusion is to prevent cycle creation in a DAG. In a DAG $G = (V, E)$, 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 .

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 the graph partition, where $E(g_d)$ is the set of edges 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 $\langle i, j \rangle, \forall i, j \in V$. 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 that 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. \square



(a) Fuse D into B



(b) Fuse C into B

Figure 4.2: In (a), merging operators B and D does not increase training latency, as operators B and D are not parallelizable. Moreover, the fusion preserves the parallelization opportunity between B and C. In (b), each dashed ellipse represents a pair of parallelizable operators. Merging operators C and B where the out-degree of source C and the in-degree of destination B are both at least two removes parallelization opportunities.

Algorithm 1 Operator fusion

Input: Edge (i, j) ; computation graph G ; merging threshold α ;

Output: Boolean indicating if j will merge into i ;

```

1: if  $G.out\_degree(i) \geq 2 \wedge G.in\_degree(j) \geq 2$  then
2:   return False
3: end if
4: if  $G.out\_degree(i) = 1 \wedge G.in\_degree(j) = 1$  then
5:   return True
6: end if
7: if  $(p_i \leq \alpha \wedge G.out\_degree(i) = 1) \vee$ 
8:  $(p_j \leq \alpha \wedge G.in\_degree(j) = 1)$  then
9:   return True
10: end if
11: return False

```

Although 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 with a computation complexity of $O(1)$ to

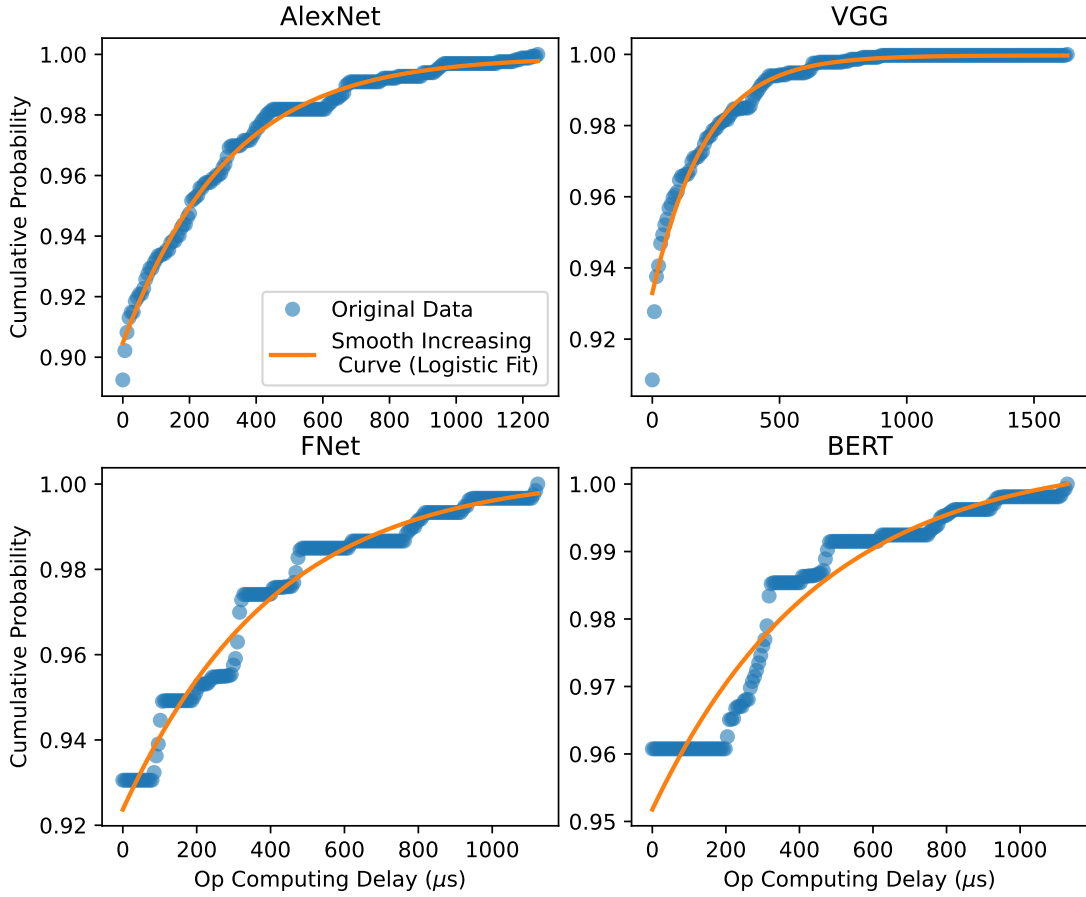


Figure 4.3: For each DNN model, the blue curve represents the CDF of operator computing delay.

determine whether operator j should fuse into i for each edge $(i, j) \in E$ iteratively, based on the following conditions.

- 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. 4.2(b), the fusion of operator C into operator B causes a significant loss of parallelization opportunities. Additionally, edges (C, B) , (C, D) , and (C, F) are removed while two new edges (BC, D) and (BC, F) are 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. 4.2(a), edge (B, D) forms a straight line-like structure because the out-degree of operator B and the in-degree of operator D are both one. By Theorem

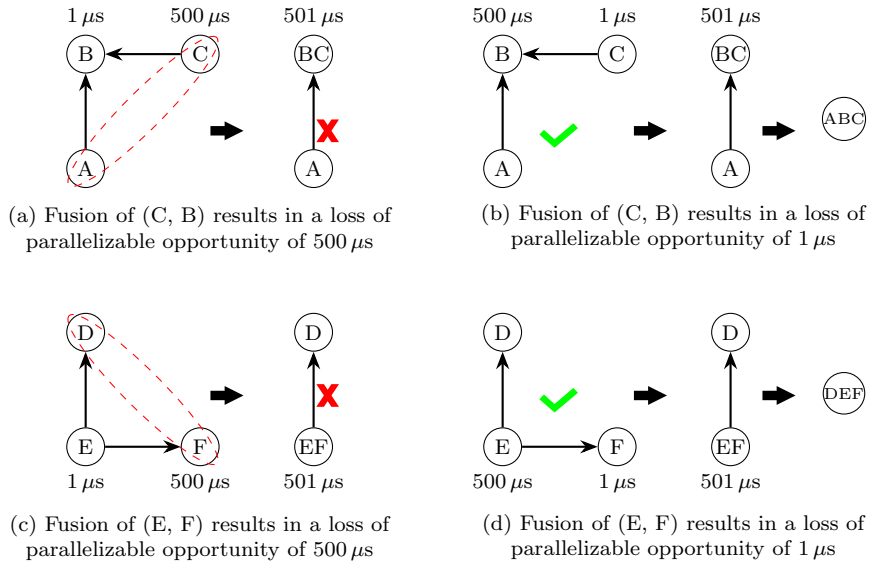


Figure 4.4: Each dashed ellipse represents a pair of parallelizable operators. (a) and (c) show that fusing an edge may cause significant performance loss if the destination operator’s in-degree or the source operator’s out-degree is at least two, even if the operator has low computing delay.

1, operators B and D are not parallelizable, since the straight-line path $B \rightsquigarrow D$ has only one possible topological sort. However, because B retains parallelization opportunities with C upon fusion, D can be fused into B without performance degradation.

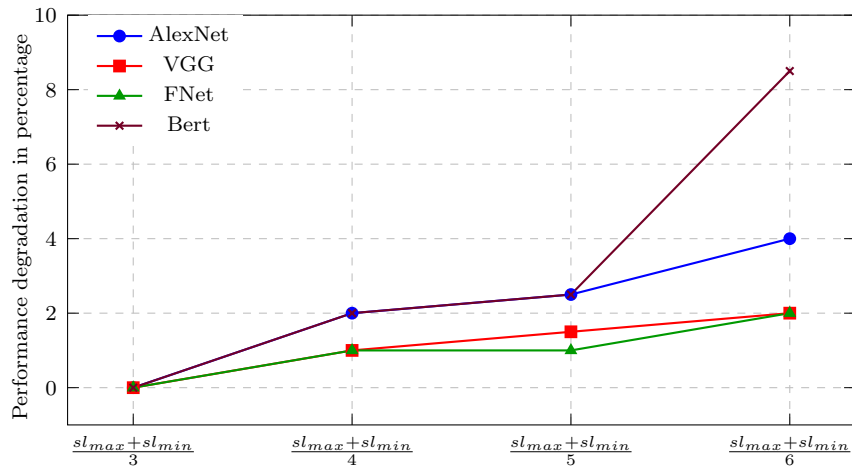


Figure 4.5: The figure demonstrates the performance degradation in training latency increases with the value of α .

- The analysis of the profiling results shows that the majority of computing demand comes from a few operators, which aligns with the findings of Pesto [16]. As shown in Fig. 4.3, the smoothed cumulative distribution function (CDF) curves of operator computing delays across different DNN models demonstrate a diminishing acceleration. We define α as the predefined value of the merging threshold. 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. 4.4(b) and Fig. 4.4(d). The value of α is carefully determined to balance the solution search latency and training latency. As shown in Fig 4.5, performance degradation in training latency increases with the value of α because a higher value of α prunes more parallelizable operator pairs while reducing the computational complexity of the optimization problem. The value of α for each model is set as $\frac{sl_{max}+sl_{min}}{4}$, where sl_{max} and sl_{min} denote the slopes at the beginning and end, respectively, of the corresponding smoothed CDF.

4.2 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 in (4.1) to calculate the 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

$$r_i = p_i + \max_{j \in \text{succ}(i)} \left(r_j + \frac{t_{i,j}}{b} \right). \quad (4.1)$$

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, $\max_{j \in \text{succ}(i)} \left(r_j + \frac{t_{i,j}}{b} \right)$, where $\text{succ}(i)$ 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 $U \in W$.

4.3 Iterative Fusion and Colocation

Once a parallelizable operator pair $\langle i, j \rangle \in R$ is placed in the same WCC U , that is, $\exists U \in W, \{i, j\} \subseteq U$, the parallelization opportunity between operators i and j is lost, as their computing periods do not overlap. Consequently, edges within each WCC $U \in W$ can be fused to further reduce the search space. However, since every operator with at least two successors will group with the successor on the critical path, fusing this successor introduces a dependency from the merged successor to other successors, as illustrated in Fig 4.6. This results in a performance degradation in training latency. For each WCC $U \in W$, we sequentially examine each edge (i, j) . Providing that the fusion of edge (i, j) does not introduce a cycle, the fusion is performed if all immediate successors of i belong to the same group U such that $succ(i) \subseteq U$, or the computing delay of j is less than α such that $p_j < \alpha$. As illustrated in Fig. 4.7, after performing fusion within WCCs, Algorithm 1 is executed again to further reduce the number of operators while minimizing the performance loss in training latency.

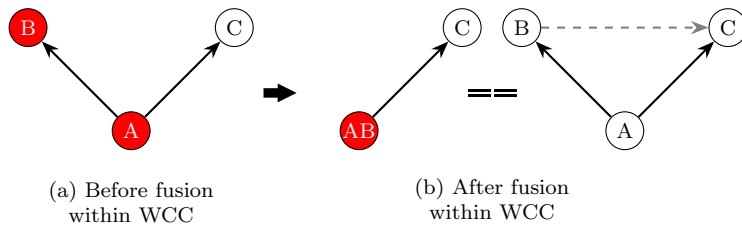


Figure 4.6: Red nodes denote operators in the same WCC co-location group. Edge fusion within a co-location group may introduce additional dependencies for operators outside the group, resulting in performance loss in training latency.

Algorithm 2 Iterative fusion and co-location

Input: Edge (i, j) ; computation graph G ,

```
1: while True do
2:   OperatorFusion( $G, \alpha$ )
3:   OperatorColocation( $G$ )
4:    $any\_update \leftarrow$  FusionWithinWCC( $G, \beta$ )
5:   if not  $any\_update$  then
6:     break
7:   end if
8: end while
9: function FUSIONWITHINWCC( $G, \beta$ )
10:   $update \leftarrow$  False
11:  for all  $(id, U) \in group\_ops\_map.items()$  do
12:     $G_U \leftarrow G.subgraph(U)$ 
13:     $E_U \leftarrow G_U.edges()$ 
14:    while  $E_U$  is not empty do
15:       $(i, j) \leftarrow E_U.pop()$ 
16:      if not IsFusionAcyclic( $G_U, i, j$ ) then
17:        continue
18:      end if
19:      if  $succ(i) \subseteq U \vee p_j < \alpha$  then
20:         $new\_edges, del\_edges \leftarrow G.merge(i, j)$ 
21:         $E_U \leftarrow E_U \cup \{(a, b) \in new\_edges \mid \{a, b\} \subseteq U\}$ 
22:         $E_U \leftarrow E_U \setminus del\_edges$ 
23:         $update \leftarrow$  True
24:      end if
25:    end while
26:  end for
27: end function
```

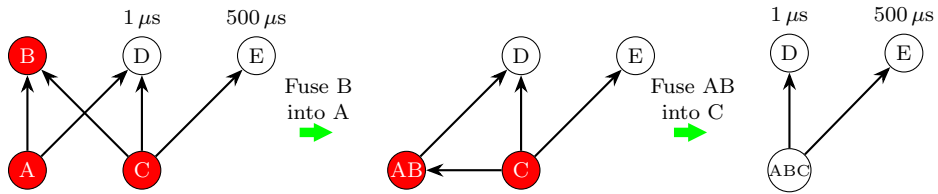


Figure 4.7: Red nodes denote operators in the same WCC group. After fusion within the WCC, Algorithm 1 can detect new fusible operator D.

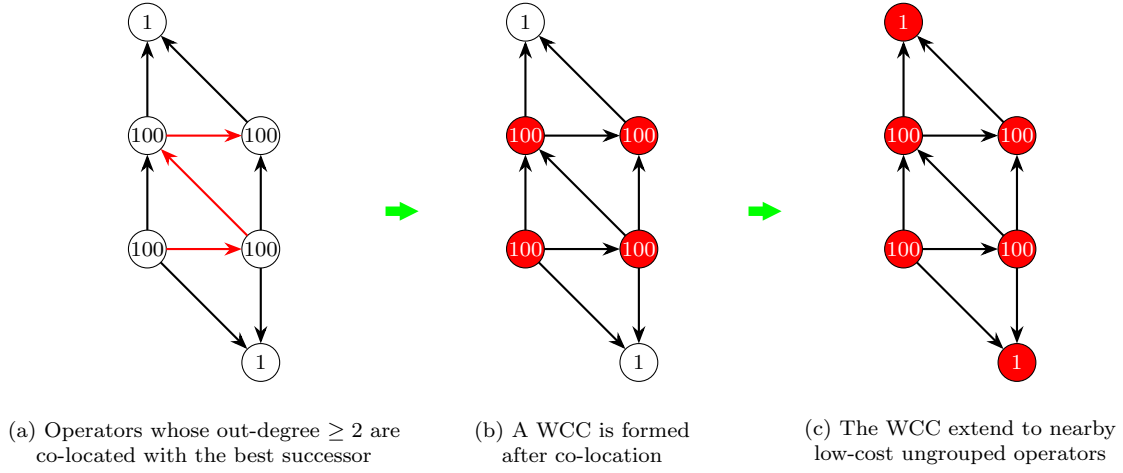


Figure 4.8: This is an example of a DAG computation graph after operator co-location. We assume that each cross-GPU communication takes five microseconds. Red nodes denote operators in the same WCC co-location group, and the number inside each node indicates the corresponding computing delay in microseconds.

WCC expansion: For any two operators $\{i, j\}$ in the same WCC $U \in W$, the computational complexity is minimized due to the absence of communication cost calculation and the known execution order, which is predetermined by the relative order of tp_i and tp_j in the topological sort. However, combining multiple WCCs into one leads to a significant increase in training latency, as all operators across the combined WCCs are constrained to run on the same device. To avoid this, only ungrouped operators can join one WCC for expansion, further reducing the number of decision variables in the optimization process. The operator i is ungrouped if it is not co-located with any other operator, that is, $i \notin \bigcup_{U \in W} U$. We define a co-location threshold β , set to $\frac{sl_{max} + sl_{min}}{2}$. Based on β , we identify the set of operators Z consisting of all ungrouped operators whose computing delays are less than β . Then, for each operator $i \in Z$, if i has an immediate successor or predecessor in a co-location group $U \in W$, i will join U , as shown in Fig 4.8.

IP reformulation: After applying iterative operator fusion and co-location, the refor-

mulated IP problem with a significantly reduced search space is presented in **P3**.

$$\mathbf{P3:} \quad \text{Min}_{\{x_{i,d}\}, \{y_{U,d}\}, \{b_{d,e}\}} A = \max_{i \in V} f_i$$

s.t.

Constraints (3.1), (3.2), (3.4), (3.7), (3.8),

$$y_{U,d} \in \{0, 1\}, \forall U \in W, \forall d \in D, \quad (4.2)$$

$$\sum_{d \in D} y_{U,d} = 1, \forall U \in W, \quad (4.3)$$

$$x_{i,d} = y_{U,d}, \forall d \in D, \forall U \in W, \forall i \in U, \quad (4.4)$$

$$\sum_{d \in D} x_{i,d} = 1, i \notin \bigcup_{U \in W} U, \quad (4.5)$$

$$\begin{cases} f_i \leq s_j, & \text{if } t_{i,j} = 0 \vee (\exists U \in W, \{i, j\} \subseteq U), \\ f_i + \sum_{\substack{d, e \in D \\ d \neq e}} \delta_{d,e}^{i,j} \cdot (p_0 + \frac{t_{i,j}}{b_{d,e}}) \leq s_j, & \text{else,} \end{cases} \forall \langle i, j \rangle \in E, \quad (4.6)$$

$$\begin{cases} f_i \leq s_j, & \text{if } \exists U \in W, \{i, j\} \subseteq U, \\ f_i \leq s_j + Q \cdot (2 - x_{i,d} - x_{j,d}), \forall d \in D, & \text{else,} \end{cases} \forall \langle i, j \rangle \in R, \quad (4.7)$$

where $y_{U,d}$ represents co-location group-device mapping indicators, and constraint (4.3) 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 (4.4). The many-to-one operator-to-device mapping applies only to ungrouped operators to reduce the complexity, as specified in the constraint (4.5). 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 (3.5), constraint (4.6) specifies that operators in the same group U do not incur communication costs because they are on the same device. $R = \{\langle i, j \rangle \in O \mid i \not\rightsquigarrow j, p_i > 0, p_j > 0\}$ represents all parallelizable operator pairs in G . By restricting the constraint (4.7) to R , its complexity is significantly reduced. For any two operators i, j belonging to the same group U , operators i and j are executed sequentially, even if $\langle i, j \rangle$ is an element of the set of parallelizable operator pairs R . Their execution order is determined by the relative order of tp_i and tp_j in the topological sort TP , such that

$tp_i < tp_j \implies f_i < s_j, \forall \{i, j\} \subseteq U, \forall U \in W$. Consequently, within each group U , the relative execution order of operators $i \in U$ follows an increasing sequence based on their topological order tp_i , thus further reducing computational complexity. Consequently, constraint (4.6) can be reformulated as,

$$\forall U \in W, \{i, j\} \not\subseteq U \implies \begin{cases} f_i \leq s_j, & \text{if } t_{i,j} = 0, \\ f_i + \sum_{\substack{d,e \in D \\ d \neq e}} \delta_{d,e}^{i,j} \cdot (p_0 + \frac{t_{i,j}}{b_{d,e}}) \leq s_j, & \text{else.} \end{cases} \quad (4.8)$$

With the much reduced number of decision variables by operator co-location, the commercial solver, i.e., Gurobi [41], is then used to optimize device placement and network bandwidth allocation decisions. Then, the per-iteration training latency is obtained. As shown in Table 5.1, for a cluster comprising four devices, **P3** requires less than 80 seconds to determine device placement decisions and network bandwidth allocation for all models because the proposed iteration operator fusion and co-location scheme significantly reduces the search space for device placement decisions and network bandwidth allocation optimization. This performance demonstrates its practical efficiency in solving NP-hard problems. Meanwhile, as demonstrated in Fig. 5.4, the proposed iterative operator fusion and co-location scheme incurs a performance loss of up to 7 % in per-iteration latency compared to the near-optimal one.

Chapter 5

Performance Evaluation

In this section, we first describe the implementation, experimental setup, models, dataset, and baseline approaches for device placement search, search space reduction, and network topology. Subsequently, we present experiments on network bandwidth allocation and simulation results for parallelization, and then combine these components for comprehensive emulation. It first validates the effectiveness of the proposed network bandwidth allocation scheme and then quantifies the per-iteration training latency and solution search latency across different GPU counts. Finally, it evaluates the benefits of network bandwidth allocation under diverse link bandwidth configurations.

5.1 Experimental Setup

Testbed Setup: The cluster consists of two servers and two network switches. Each server is equipped with an Intel Core i7-12700K CPU and an NVIDIA GeForce RTX-4080 super GPU to provide substantial computational and graphics processing capabilities. Link redundancy and bandwidth are ensured through a dual-port NIC installed in each server; each NIC port is connected to a separate upstream switch via a dedicated 1 Gbps Ethernet link. All servers run Ubuntu 22.04 LTS as the operating system, ensuring a stable and up-to-date Linux environment for high-performance computing tasks.

Simulation Setup: We perform profiling on an NVIDIA RTX-4080 super GPU. The IP optimizer, Gurobi, explores the search space once it finds a solution within 10% of the

optimum. For each model, the optimization process is repeated 20 times using the same number of devices, and we compute the average solution search latency and the expected training latency.

DNN Models and Datasets: We train image classification models, such as AlexNet [9] and VGG16 [6], on the CIFAR-10 [42] dataset, which consists of ten image classes, using a batch size of 512. For natural language processing (NLP) models, including BERT [43] and FNet [44], we utilize IMDB movie reviews from the TensorFlow dataset, which contains two classes, with a batch size of 64. To maintain a consistent input length, each text sample is padded to 128 tokens.

5.2 Implementation

The proposed solution is implemented using TensorFlow 2.16 [45]. During training, the `graph` API is used to trace and extract the computation graph, which is then modeled using NetworkX [46]. 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 more than 100,000 iterations and multiple epochs, this incurs a profiling overhead of less than 0.1% [16]. The Tensorflow profiler calculates the computing delay and memory cost of the operators, which serve as input to the MIP model implemented by the commercial solver Gurobi [41]. We evaluated all baseline placement search strategies using the simulator proposed by FlexFlow [13]. As the simulator employs a default first-in-first-out scheduling policy, which is suboptimal, we adopt the list scheduling scheme proposed in HeteroG [36] to obtain more accurate and representative evaluation results for the baseline strategies.

5.3 Baseline

Device placement searching strategies:

- Metis [30, 31, 17] is an open-source tool for graph partitioning. It assigns computing delay as the operator weight and communication cost as the edge weight. By balancing the total operator weights across subgraphs while minimizing the edge cut weight, Metis ensures even load distribution and reduces communication overhead.
- MCMC [13, 23] employs an iterative approach to optimize operator placement. It starts with an initial placement strategy and randomly modifies the assigned device for an operator. If the new placement results in a lower cost based on the cost model, it replaces the current strategy, enabling efficient exploration of the search space. The initial placement strategy significantly impacts the performance of MCMC. To prevent training latency from exceeding that of single-device placement, we initialize with a single-device configuration instead of a randomly generated one in FlexFlow [13]. This significantly improves the performance of MCMC.

Search space reduction schemes:

- ColocRL [19]: An operator is co-located with its successor if and only if its out-degree is equal to one. The heuristic is applied recursively until no further groups can be formed.
- PlaceTo [18]: The algorithm iteratively merges the operator with the lowest cost into one of its successor operators. If the selected operator has no successors, it is instead merged into one of its predecessors. This process continues until the computation graph is reduced to a target size \mathcal{N} , which is set to match the size of the computation graph produced by the proposed iterative operator fusion and co-location scheme, as reported in Table 5.1.
- SpotLight [34]: Operations are grouped into a super operator if they share the same two-level prefix. For example, `sequential_1/conv2d_1/convolution` and `sequential_1/conv2d_1/Reshape` share the same two-level prefix `sequential_1/conv2d_1`, and are therefore grouped into the same super operator.

Simulated network architectures:

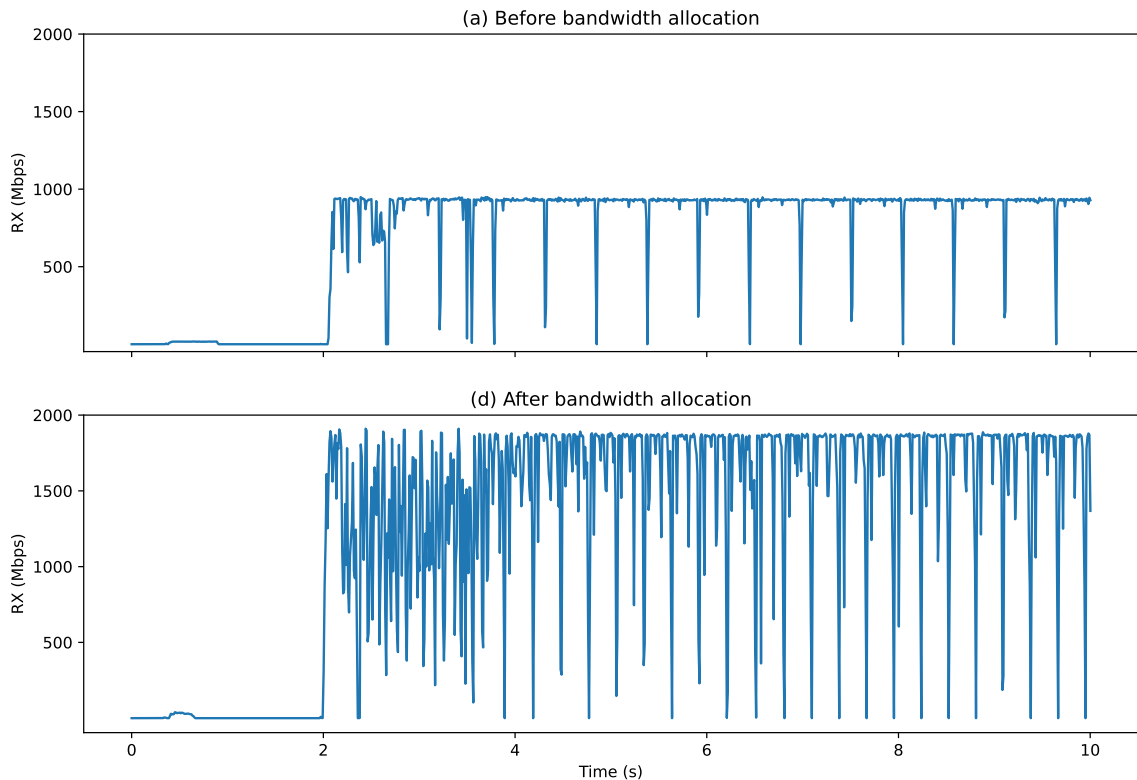


Figure 5.1: The figure shows the comparison of incoming bandwidth utilization on the network interfaces of one server during the first 10 seconds of distributed data parallel training with VGG16, before and after bandwidth allocation.

- **Non-blocking tree:** To evaluate the impact of bandwidth allocation optimization on per-iteration training latency, we simulate a full-bisection bandwidth tree topology comprising four switches, with no oversubscription. The problem formulation is the same as **P1**. In this setup, each server is equipped with a single network interface card (NIC) and is capable of communicating with any other server at the full bandwidth denoted by b .

5.4 Result Analysis

Bandwidth optimization: The experimental results in Fig. 5.1 show the incoming bandwidth utilization on the network interfaces of one server during distributed data parallel training, before and after bandwidth allocation. Compared with Fig. 5.1(a), the

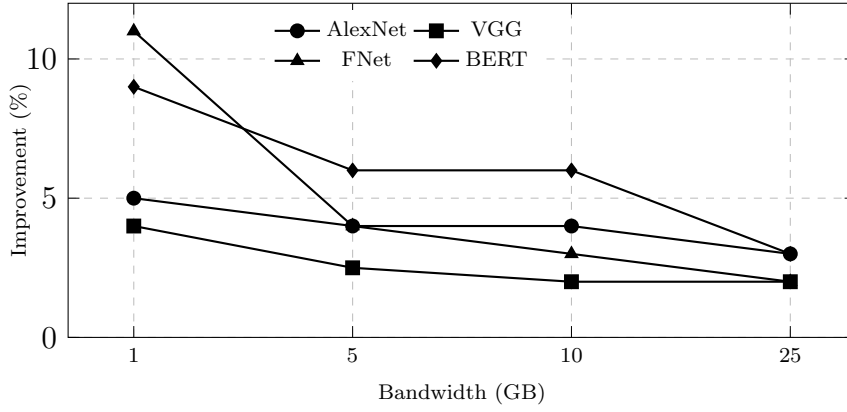


Figure 5.2: This figure illustrates the improvement in per-iteration training latency achieved through bandwidth optimization, compared to the non-blocking tree-based baseline, in a setup involving four servers across different link bandwidth.

narrower pattern in Fig. 5.1(b) indicates that by utilizing the bandwidth of both network links, all-reduce communication requires less time to synchronize gradients for each training step. Furthermore, the simulation results in Fig. 5.2 demonstrate that bandwidth allocation optimization consistently reduces per-iteration training latency across all link bandwidth scenarios. However, the extent of improvement diminishes as link bandwidth increases. This is because communication constitutes the primary bottleneck, whereas the bottleneck shifts toward computation as the available bandwidth increases.

Placement search latency: Table 5.1 compares the placement search latency of our solution with other baseline strategies. As presented in Table 5.1, the placement search latency in our solution scales approximately linearly with the number of operators because solver runtime is primarily influenced by the number of parallelizable operator pairs, and our proposed iterative operator fusion and co-location scheme efficiently prunes the search space by eliminating parallelizable operator pairs that contribute negligibly to training latency reduction. However, when device and bandwidth allocations are co-optimized, the solver runtime exhibits exponential growth with respect to the number of operators. In contrast, the graph partitioning latency of Metis scales linearly with the number of operators but remains unaffected by the number of GPUs, as it relies on multilevel coarsening to perform the initial K-way cut on a small graph.

Table 5.1: Average placement search latency comparison in seconds

Model	# of Operators (Pre \rightarrow Post Fusion)	# of WCC Groups	Operator Fusion Algo. Runtime	Solver Runtime of Our Solution on Four Servers	
				Bf. Band Opt.	Af. Band Opt.
AlexNet	1,656 \rightarrow 120	13	0.1	0.7	1.0
VGG-16	3,652 \rightarrow 206	16	0.1	0.8	1.5
FNet	7,186 \rightarrow 473	61	0.4	8.3	36.4
BERT	12,716 \rightarrow 580	64	1.2	15.8	76.7

Model	Runtime of MCMC on Four Servers			Runtime of Metis on Four Servers
	5,000 Steps	15,000 Steps	25,000 Steps	
AlexNet	71	213	357	0.2
VGG-16	220	662	1,016	0.6
FNet	716	2,148	3,654	1.6
BERT	2,740	8,220	13,680	4.2

The performance of MCMC in minimizing training latency improves with an increasing number of steps, but shows diminishing returns over time. As illustrated in Fig. 5.3, while MCMC is effective for models with relatively small search spaces, the time and number of steps required to identify optimal placement decisions increase exponentially as the search space expands. Starting from an initial single-device placement, MCMC completes 25,000 steps in 6 minutes and 18 minutes for AlexNet and VGG, respectively, achieving convergence and per-iteration training latency comparable to that of our proposed solution. In contrast, for models with larger and more complex architectures, such as FNet and BERT, MCMC fails to discover effective placement strategies with comparable performance, even after 1 hour and 3 hours of execution, respectively.

Expected per-iteration training latency: The simulation results in Fig. 5.3 for four DNN models show that our solution consistently outperforms all baseline strategies and achieves up to a 22% 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, subgraphs on different devices tend to develop stronger inter-dependencies, leading to device idle time while waiting for cross-device dependencies and ultimately

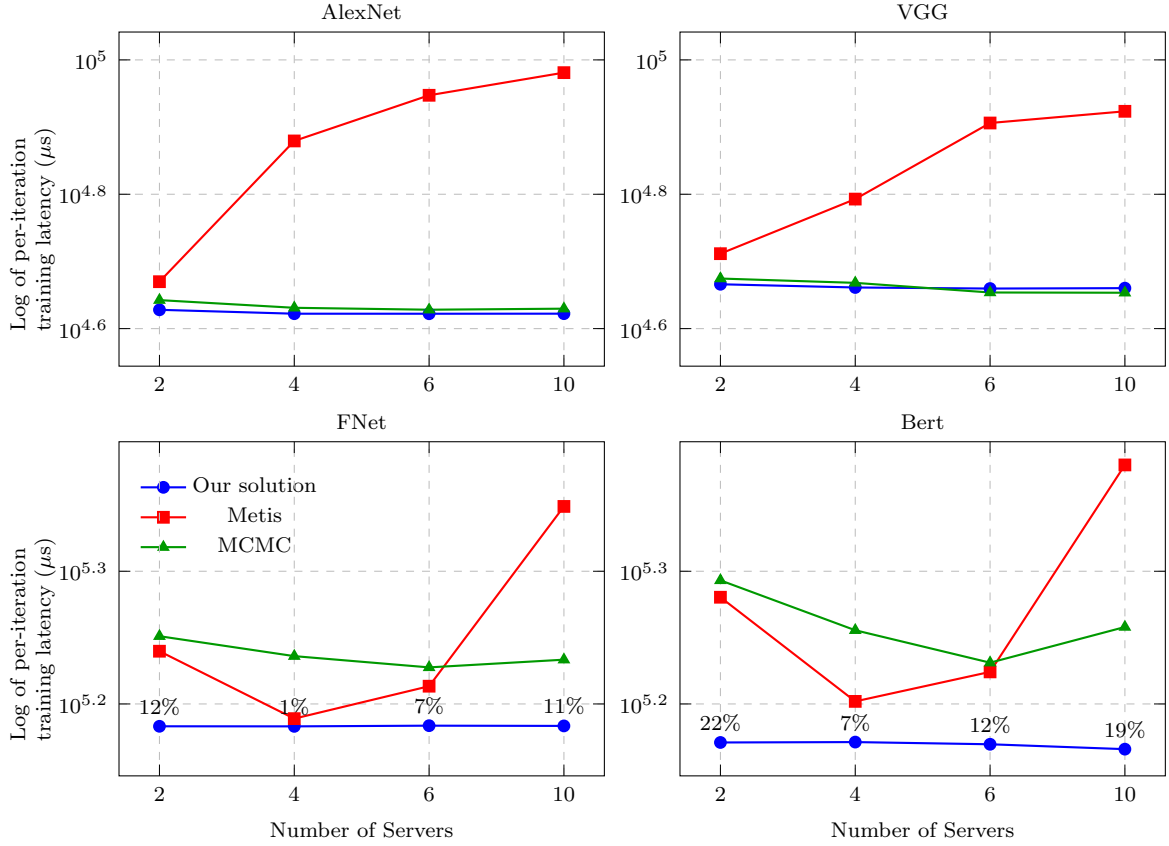


Figure 5.3: This figure compares the per-iteration training latency for two baseline strategies and our proposed method without bandwidth allocation optimization. The number above each circle indicates the performance improvement of our solution over the best alternative strategy. MCMC per-iteration training latency is recorded after 25,000 steps.

limiting scalability.

Performance loss analysis: The proposed iterative operator fusion and co-location scheme significantly reduces the search space caused by tens of thousands of operators in the computation graph, with minimal performance degradation in training latency. As illustrated in Fig. 5.4, the per-iteration training latency of our approach consistently outperforms the algebraic transformation schemes proposed in PlaceTo [18] and SpotLight [34], although incurring a performance loss of up to 7 % compared to ColocRL [19]. Furthermore, as demonstrated in Table 5.2, our method achieves the lowest placement search latency among all baselines. For example, our design is up to 165 times faster than ColocRL. This efficiency gain arises because the baselines leverage only operator fusion or co-location and fail to effectively prune the search space for parallelizable operators

Table 5.2: Comparison of placement search latency under different search space reduction schemes in seconds

Model	Solver Runtime among Transformation Schemes							
	Our solution		PlaceTo		SpotLight		ColocRL	
	2	4	2	4	2	4	2	4
AlexNet	0.2	0.7	1.1	2.38	36.6	178.7	116.5	OOM
VGG-16	0.2	0.8	1.9	16.19	8.6	42.7	43.4	103.7
FNet	6.9	8.3	54.4	OOM	3.1	4.6	71.4	OOM
BERT	17.7	15.8	292.9	OOM	2.7	11.4	OOM	OOM

contributing minimally to training latency reduction. Additionally, the pruned search space reduces memory overhead, mitigating the risk of out-of-memory (OOM) errors. This is critical because solvers such as Gurobi store large sparse matrices and related data structures for variables and constraints in RAM to solve the problem efficiently.

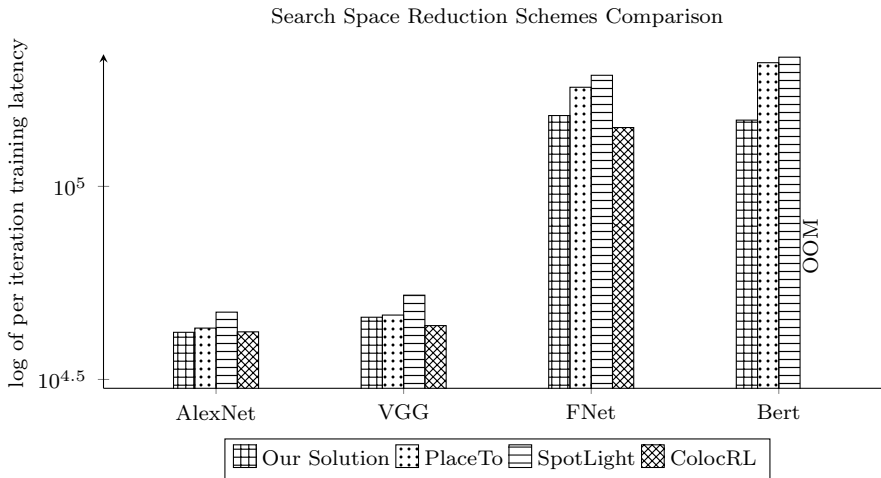


Figure 5.4: This figure compares training latency on logarithmic scale across different search space reduction schemes on two devices. OOM indicates an out-of-memory error.

Chapter 6

Conclusions and Future Works

In this thesis, we have investigated the problem of joint optimization of device placement, algebraic transformation, and network bandwidth allocation to accelerate large-scale distributed DNN training. As discussed in Chapter 3, to tackle the challenge of low network bandwidth resource utilization in Clos-based network architectures, we have proposed a network bandwidth allocation scheme to allocate bandwidth resources asymmetrically across inter-server communication pairs. This approach accommodates the non-uniform traffic patterns inherent in DNN training workloads. If bandwidth demand from a source server exceeds the capacity of a single link to the destination server, this approach utilizes the bandwidth resources of multiple links simultaneously, thus reducing the communication cost and enabling our approach to outperform the tradition Clos-based topology whose maximum bandwidth achievable does not exceed single link. To tackle the challenges of NP-hardness and ever-increasing model sizes, we have presented an algebraic transformation scheme based on iterative operator fusion and co-location to reduce the search space generated by device placement and network bandwidth allocation optimization while incurring minimal subsequent performance loss in training latency. 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 22% reduction in per-iteration training latency. Incorporating additional net-

work bandwidth allocation can provide up to an additional 11% reduction in training latency. More importantly, our design achieves up to 650 times lower solution search latency compared with SOTA methods.

Distributed DNN training involves data parallelism and various forms of model parallelism. For future work, the computation graph and the system model can be modified to support data and tensor parallelism by duplicating or splitting operators. The optimization problem can determine the optimal parallelization strategies by jointly considering all these approaches. This allows the device to further improve device utilization, reducing training latency. Meanwhile, InfiniBand, NVLink, and UB-Mesh represent a new generation of inter-server communication technologies, offering significantly higher bandwidth and substantially lower communication setup latency. By incorporating link aggregation, the proposed network bandwidth allocation scheme can be extended to accommodate servers interconnected using these advanced communication methods. This can further reduce the communication overhead and enhance the scalability of large-scale DNN training. Finally, this device placement search can be adapted to a heterogeneous environment where each operator experiences varying computing delays on different devices.

Appendix A

Selected Publications

The following publication reflects the outcome of my research and collaboration, as the main author, in partnership with my supervisor, and lab colleagues during my Master’s program at Memorial University of Newfoundland.

- Hong Wang, Jinhao Luo, Kaiyang Liu, and Qiang (John) Ye. Efficient Device Placement for Distributed DNN Training. In *IEEE ICC*, Montreal, Canada, 2025, accepted for publication.

First, I developed a search space reduction scheme based on operator fusion and co-location to address the NP-hardness of the device placement decision search problem and increased DNN model sizes while minimizing the resulting performance degradation in training latency. This work has been published in *IEEE ICC 2025*. Building upon this, the thesis proposes an iterative operator fusion and co-location approach that further reduces solution search latency. Additionally, by considering a computing cluster similar to that studied in *TopoOpt*, the proposed method jointly optimizes device placement decisions and network bandwidth allocation, effectively addressing the challenge of low network resource utilization caused by non-uniform traffic distribution in traditional Clos-based architectures for distributed DNN training.

Bibliography

- [1] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.
- [2] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *NC*, 9(8):1735–1780, 11 1997.
- [3] Junyoung Chung, Çağlar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.
- [4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *NeurIPS*, volume 30. Curran Associates, Inc., 2017.
- [5] Jiahui Yu, Zirui Wang, Vijay Vasudevan, Legg Yeung, Mojtaba Seyedhosseini, and Yonghui Wu. Coca: Contrastive captioners are image-text foundation models, 2022.
- [6] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.
- [7] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, and Aiesha Letman. The Llama 3 herd of models, 2024.
- [8] Chuan Li. Demystifying GPT-3, 2020.
- [9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, 2017.

- [10] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *NeurIPS*, volume 32, pages 102–113. Curran Associates, Inc., 2019.
- [11] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. PipeDream: Fast and efficient pipeline parallel DNN training, 2018.
- [12] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *SC*, page 1–14. ACM, November 2021.
- [13] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *MLSys*, volume 1, pages 1–13, 2019.
- [14] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism, 2020.
- [15] Lianmin Zheng, Zhuohan Li, Haichen Zhang, Yida Zhuang, Zhihao Chen, and Yuke Huang. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *USENIX OSDI*, pages 559–578, 2022.
- [16] Ubaid Ullah Hafeez, Xiao Sun, Anshul Gandhi, and Zhenhua Liu. Towards optimal placement and scheduling of DNN operations with pesto. In *Middleware*, pages 39–51, 2021.
- [17] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Reinforced genetic algorithm learning for optimizing computation graphs, 2020.
- [18] Ravichandra Addanki, Shaileshh Bojja Venkatakkrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. Placeto: Learning generalizable device placement algorithms for distributed machine learning. In *NeurIPS*. Curran Associates Inc., 2019.

- [19] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In Doina Precup and Yee Whye Teh, editors, *ICML*, volume 70, pages 2430–2439, 06–11 Aug 2017.
- [20] Shiwei Zhang, Xiaodong Yi, Lansong Diao, Chuan Wu, Siyu Wang, and Wei Lin. Expediting distributed DNN training with device topology-aware graph deployment. *IEEE TPDS*, 34(4):1281–1293, 2023.
- [21] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *USENIX OSDI*, pages 267–284, Carlsbad, CA, July 2022. USENIX Association.
- [22] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM*, page 63–74, 2008.
- [23] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Zhijao Jia, Dheevatsa Mudigere, Ying Zhang, Anthony Kewitsch, and Manya Ghobadi. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *USENIX NSDI*, pages 739–767, 2023.
- [24] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The design and implementation of open vSwitch. In *USENIX NSDI*, page 117–130, 2015.
- [25] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based load balancing for fast datacenter networks. In *ACM SIGCOMM*, page 465–478, New York, NY, USA, 2015.

- [26] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2):69–74, 2008.
- [27] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In *NeurIPS*, page 1223–1231. Curran Associates, Inc., 2012.
- [28] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.
- [29] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.
- [30] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.
- [31] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C. Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, and James Laudon. GDP: Generalized device placement for dataflow graphs, 2019.
- [32] Wil Michiels, Emile Aarts, and Jan Korst. *Theoretical Aspects of Local Search (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag, Berlin, Heidelberg, 2007.
- [33] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. A hierarchical model for device placement. In *ICLR*, 2018.
- [34] Yuanxiang Gao, Li Chen, and Baochun Li. Spotlight: Optimizing device placement for training deep neural networks. In Jennifer Dy and Andreas Krause, editors, *ICML*, volume 80, pages 1676–1684. PMLR, 10–15 Jul 2018.
- [35] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter Ma, Qiumin Xu, Hanxiao Liu, Phitchaya Mangpo Phothilimthana, Shen Wang, Anna Goldie, Azalia

- Mirhoseini, and James Laudon. Transferable graph optimizers for ML compilers, 2021.
- [36] Xiaodong Yi, Shiwei Zhang, Ziyue Luo, Guoping Long, Lansong Diao, Chuan Wu, Zhen Zheng, Jun Yang, and Wei Lin. Optimizing distributed training deployment in heterogeneous GPU clusters. In *ACM CoNEXT*, pages 93–107, 2020.
- [37] Jakub Tarnawski, Amar Phanishayee, Nikhil R. Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of DNN graph operators. In *NeurIPS*, volume 33, pages 15451–15463, 2020.
- [38] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *IEEE ISPASS*, page 283–294, April 2023.
- [39] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. In *USENIX OSDI*, pages 481–498, 2020.
- [40] Arthur B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962.
- [41] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024.
- [42] Alex Krizhevsky. Learning multiple layers of features from tiny images, 2009.
- [43] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.
- [44] James Lee-Thorp, Joshua Ainslie, Ilya Eckstein, and Santiago Ontaño. FNet: Mixing tokens with fourier transforms. *CoRR*, abs/2105.03824, 2021.
- [45] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, and J. Dean. TensorFlow: A system for large-scale machine learning. In *USENIX OSDI*, pages 265–283, 2016.

- [46] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *SciPy*, pages 11–15, 2008.