Rethinking Dynamic Networks and Heterogeneous Computing with Automatic Parallelization

Ruilong WU
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
rwu408@connect.hkust-gz.edu.cn

Xinjiao Li
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
xli886@connect.hkust-gz.edu.cn

Yisu Wang
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
ywang418@connect.hkust-gz.edu.cn

Xinyu Chen
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
xinyuchen@hkust-gz.edu.cn

Dirk Kutscher
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
dku@hkust-gz.edu.cn

Abstract

Hybrid parallelism techniques are essential for efficiently training large language models (LLMs). Nevertheless, current automatic parallel planning frameworks often overlook the simultaneous consideration of node heterogeneity and dynamic network topology changes, limiting their effectiveness in practical applications. In this paper, we address these limitations by modeling heterogeneous nodes within dynamically changing network environments and leveraging simulation-based strategies to determine optimal parallel configurations. Our approach enables fine-grained workload allocation tailored for heterogeneous nodes and complex network scenarios, achieving performance competitive with state-of-the-art methods under regular and stable network conditions. Additionally, we introduce a strategy pruning technique to rapidly discard infeasible parallel configurations, substantially reducing the search space and accelerating the search process through parallel execution within the simulator. Preliminary evaluations confirm that our method notably enhances training performance on heterogeneous nodes and demonstrates improved adaptability in complex, dynamic scenarios such as cloud computing environments.

CCS Concepts

• **Networks → Data center networks**.

Keywords

Dynamic Networks, Hybrid parallelism, Distributed Training, Heterogeneous Computing

1 Introduction

The rapid growth in parameter count of deep neural networks (DNNs), especially large language models (LLMs)[30][2][34][29][12][24] based on the Transformer[36] architecture, has made distributed parallel training across large GPU clusters indispensable. Thus, efficient implementation of distributed training is critical. Researchers have proposed various parallel strategies[27][31][32], including Data Parallelism (DP)[23], Tensor Parallelism (TP)[27], Pipeline Parallelism (PP)[14][9][22][25][10], Sequence Parallelism (SP)[19], and Fully Sharded Data Parallelism (FSDP)[41], to address computational, storage, and communication challenges in training large models. However, selecting appropriate parallel strategies in practical large-scale clusters typically requires extensive manual tuning. While existing automatic search frameworks, such as ALPA[42], AMP[21], Metis[35] and Galvatron[26], offer some degree of automation, their deployment in real-world scenarios is limited due to overly idealized assumptions.

This paper aims to address the problem of selecting distributed parallel strategies more effectively for realistic scenarios. Our core insight is that **computation can be viewed as mapping data and algorithms onto computational devices, while communication corresponds to data transmission tasks across network links**. Specifically, by selecting suitable parallel strategies, computational tasks can be efficiently assigned to heterogeneous computing devices and communicated through network links. However, due to device performance heterogeneity and dynamic network conditions, the actual execution time of tasks typically exhibits significant uncertainty.

The uncertainty in task execution time arises mainly from two aspects: first, variability in computation and communication times caused by heterogeneous device performance and fluctuations in network bandwidth; second, additional variations resulting from operators' splitting and fusion processes. For example, operator fusion reduces memory accesses and thus shortens execution time, while operator splitting can effectively utilize idle computational resources, also reducing execution time. Additionally, decomposing traditional collective communication operations such as `all-reduce` into `reduce-scatter` and `all-gather` can significantly enhance communication efficiency.

To address these practical challenges, we propose an integrated optimization framework combining strategic operator splitting and

fusion, an adaptive task scheduling strategy based on parallelized branch-and-bound search, and resource management strategies tailored for heterogeneous computational environments and dynamic network conditions. We validate our framework using SimAI[37], an existing performance prediction model, and demonstrate significant performance improvements over mainstream frameworks. Specifically, our contributions include:

- A novel multi-edge physical link abstraction model that more accurately describes heterogeneous device connectivity characteristics and link contention conditions;
- A parallelized branch-and-bound optimization algorithm that systematically searches task scheduling strategies, significantly improving task execution efficiency;
- Preliminary experimental validation using SimAI, indicating the potential of our method to outperform existing mainstream frameworks under heterogeneous computational environments and dynamic network conditions.

2 Background and Motivation

This section addresses critical challenges faced in realistic distributed training environments, specifically illustrated by the scenarios depicted in Figure 1. In practical GPU clusters, several factors significantly impact overall training efficiency and robustness: (1) heterogeneous GPU setups combining diverse device types, (2) unbalanced network bandwidth causing performance bottlenecks, and (3) node failures resulting in computational disruptions.

In Section 2.1, we first examine the impact of GPU performance heterogeneity on overall system throughput and discuss predictive performance modeling approaches. In Section 2.2, we analyze dynamic network conditions, emphasizing the necessity for adaptive bandwidth management and fault-tolerance mechanisms. Lastly, in Section 2.3, we explore strategic operator fusion and splitting methods, highlighting their potential to effectively mitigate performance degradation and improve resource utilization under these challenging conditions.
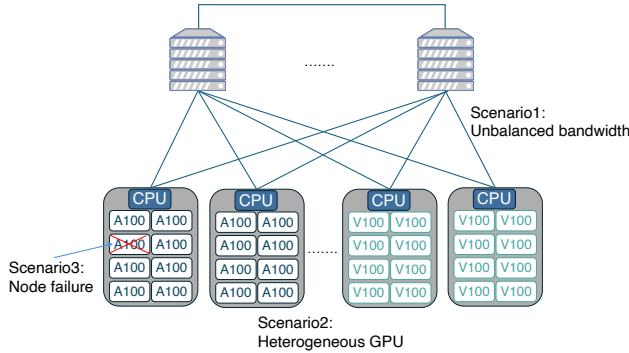

Figure 1: Representative Scenarios on hybrid parallelism: S1 Dynamic Bandwidth Variations; S2 Heterogeneous GPU Performance; S3 Device changes and connection adjustments caused by node failures

2.1 Performance Heterogeneity

Performance heterogeneity refers to variations in computational speed and capabilities among devices of the same type. Even if all nodes within a cluster employ GPUs that share the same instruction

set (e.g., CUDA), significant performance disparities can still exist due to differences in microarchitecture or hardware generation.
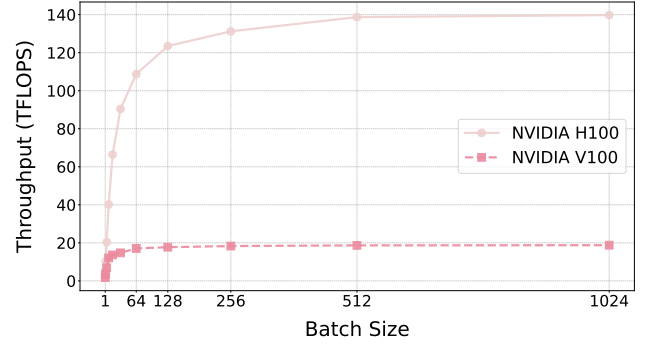

Figure 2: Attention Throughput: H100 vs V100

The Roofline Model[38] is commonly used to analyze and predict computational system performance. It characterizes the performance of a system using the following equation:

$$\text{roofline}_{BW} = \min(K \times \text{memBW}_p, \text{FLOPs}_p) \tag{1}$$

where FLOPs_p is the peak floating-point operations per second and memBW_p is the peak memory bandwidth of the GPU. The term K represents the arithmetic intensity, defined as the number of floating-point operations per memory access, computed as:

$$K = \frac{\text{FLOPs}_k}{\text{mem}_k} \tag{2}$$

Figure 2 illustrates throughput differences between H100 and V100 GPUs executing the same attention kernel. We observe significant computational capability differences between these GPUs. Once the computational load reaches a certain threshold, the GPU throughput stabilizes at a constant value.

However, the Roofline Model has limitations in accurately modeling fused operators and operations with explicitly specified resource usage, as actual GPU execution times are heavily influenced by specific hardware attributes. Prior research[40] [20] has employed Multi-Layer Perceptron (MLP) to estimate GPU performance, addressing performance as a **nonlinear**, **multivariate** function. In such scenarios, traditional optimization methods like Integer Linear Programming (ILP) and Dynamic Programming (DP) struggle to effectively map variable-length operators onto devices. This limitation arises fundamentally because ILP and DP cannot solve optimization problems with nonlinear objectives in convex spaces.

In contrast, simulators precisely predict CUDA kernel execution times, providing accurate operator execution time estimates. Moreover, simulators can concurrently evaluate execution times for multiple scheduling strategies, significantly accelerating the identification of optimal parallelization strategies.

2.2 Dynamic Networks

Dynamic networks are characterized by topological changes over time, contrasting with static networks that maintain constant nodes and edges. Formally, a dynamic network can be modeled as a temporal graph, represented by a sequence G(0), G(1), ..., G(t), or a time-dependent edge set E(t).

Efficient parallel training of large language models (LLMs) across multiple GPUs inherently faces dynamic network conditions. Communication bandwidth fluctuates due to hardware limitations or network congestion, while long-running tasks frequently experience node slowdowns or failures.

2.2.1 Dynamic Bandwidth Variations. In practical distributed training scenarios, the available bandwidth among nodes and within nodes frequently fluctuates rather than remaining constant. Such variations stem from multi-tenant data center networks, hardware bottlenecks, and background workloads. However, current distributed training frameworks[23][1] [33][17] typically cause GPUs with higher bandwidth lanes to idle, waiting for GPUs with lower bandwidth lanes to complete data transmission, despite similar computational capabilities.

2.2.2 Dynamic Node and Interconnect Adjustments. In long-running, large-scale training tasks, node failures or temporary disconnections are inevitable. Traditional approaches typically halt training upon encountering node failures, reloading from checkpoints, and restarting new nodes, resulting in substantial downtime and wastage of computational resources. Recent research has emphasized fault-tolerance capabilities, which enable distributed training systems to operate continuously despite node additions or removals. For instance, ReCycle[11] leverages the redundancy inherent in data-parallel training by dynamically reallocating workloads from failed nodes to the remaining active nodes, avoiding delays from node replacement. Oobleck[15] proactively computes pipeline-parallel configurations optimized for varying numbers of nodes, seamlessly transitioning to smaller-scale configurations upon node removal, thereby eliminating the need for retraining.

2.3 Operation Fusion and Split

Modern machine learning frameworks [23][41][3][16] typically accelerate computation by forming efficient fused kernels through the fusion of multiple consecutive operators, thereby reducing data movements from external memory. A classic example is *Flash Attention*[8], which combines originally independent operations such as `matmul`, `dropout`, `softmax`, and `mask` into a single fused kernel, significantly shortening execution time.

In contrast to operator fusion, distributed computations often utilize an operator-splitting strategy, exemplified by decomposing the standard `All-Reduce` operation into two sub-operations: `Reduce-Scatter` and `All-Gather`. As illustrated in Figure 3, the traditional `All-Reduce` aggregates gradients fully at a single node before broadcasting the result to all other nodes. The decomposed approach, however, first partitions and aggregates gradients across nodes via the `Reduce-Scatter` step, and subsequently disseminates these partial aggregation results to all nodes through the `All-Gather` step. This decomposition effectively eliminates single-node bottlenecks and enhances overall communication efficiency.

Moreover, the process of splitting and recombining operators introduces additional opportunities for optimization in parallel computation. By decomposing operators into smaller sub-operations and recombining them in novel configurations, new operators with varying execution characteristics emerge. Searching through these configurations enables identification of optimal mappings onto heterogeneous hardware, thus effectively mitigating the previously
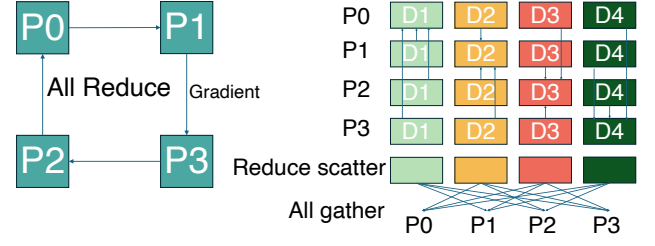

Figure 3: Comparison between traditional All-Reduce and decomposed All-Reduce (Reduce-Scatter followed by All-Gather).

discussed straggler effect, and ultimately leading to more balanced workload distribution and improved overall performance.

3 Design

We extend the traditional optimization approach of tensor programs[16][39][13] to automatic parallel strategies across devices. Our core idea is to split operators into lower-level sub-operators and then recombine them into new operators using a simulator to predict their execution times. Constraints such as data dependencies, memory size, and bandwidth are considered to identify the optimal parallel strategy. Unlike previous methods that focused purely on parallel strategy search or heterogeneous computing, our method bridges the gap between tensor-program optimization and model parallelism. As illustrated in Figure 4, GPU devices possess four distinct memory hierarchy levels, each of which offers different bandwidth characteristics. The placement of data at these varying levels directly influences the computational efficiency. Given the heterogeneous environment and dynamic network conditions, operator execution times vary across different devices and their interconnections, and this variation does not adhere to simple linear relationships. Consequently, our approach—splitting operators first and then fusing them—allows the discovery of superior strategies. Although our method could theoretically support deeper hierarchical optimizations, our current work focuses only on first-level optimization, specifically, splitting models across different devices and searching at the global memory level to obtain optimal parallel strategies.

In Section 3.1, we introduce a Multi-Edges Assumption, and in Section 3.2, we introduce the Problem Formulation. In Section 3.3, the proposed algorithm is discussed. In Section 3.4, we discuss the search space used in our design.

3.1 Multi-Edges Assumption

The introduction of multi-edges is motivated by the fact that in real-world scenarios, a single device often has multiple physical links to other devices. As illustrated in Figure 5(a), within a DGX H100[7] system, the connections from each GPU to the NVSwitch are uneven, and the NVSwitches located on both sides have more ports and higher bandwidth connections, indicating that modeling these interconnected paths as equivalent connections may lead to significant discrepancies in transmission time. In addition, in NVIDIA DGX servers[7][5][6][4], the NVSwitch can perform simple arithmetic operations, which may reduce transmission bandwidth and
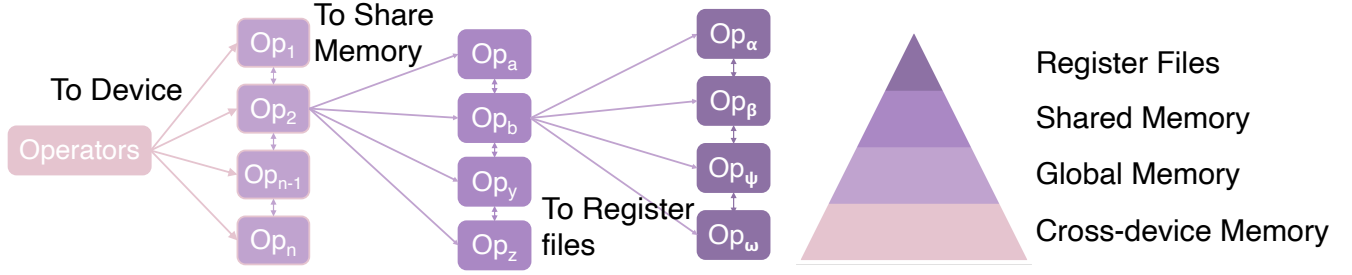
Figure 4: Hierarchy of GPU memory bandwidth optimization levels. The first level represents inter-device connections, providing a bandwidth ranging from several GB/s to tens of GB/s. The second level indicates global memory, with a bandwidth typically ranging from hundreds of GB/s to approximately 1 TB/s. The third level corresponds to the shared memory, which offers a bandwidth of several TB/s. Finally, the highest level is the register file, delivering the greatest bandwidth, typically tens of TB/s.

thus influence transmission times. Similarly, Google's TPU[18][28] employs a torus/mesh architecture that provides multiple interconnected paths across different dimensions. Moreover, as shown in Figure 5(b), typical NVIDIA GPUs offer NVLink C2C and PCIe connections simultaneously. Although the cudaMemcpy function defaults to using the NVLink connection unless explicitly disabled by invoking cudaDeviceDisablePeerAccess, for NVIDIA GPUs, the NVLink and PCIe transfers cannot be simultaneously activated within the same kernel execution. Given these considerations, introducing a multi-edge design to explicitly model multiple physical links as potentially concurrent or conflicting network resources is essential. This approach accurately simulates parallel transmissions, avoids overly simplistic single-bandwidth assumptions, and properly manages link contention states during data-transfer scheduling.
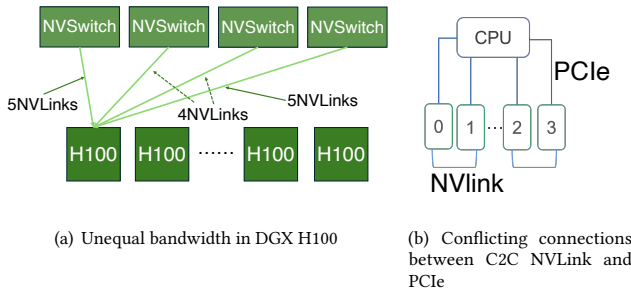

(a) Unequal bandwidth in DGX H100   (b) Conflicting connections between C2C NVLink and PCIe

Figure 5: Two typical types of links: (a) unequal bandwidth, (b) conflicting connections.

3.2 Problem Formulation

We formulate an operator-level scheduling and resource allocation problem for distributed DNN tasks across a heterogeneous multi-edge device graph. The goal is to optimally schedule computational operators considering their data dependencies and recombination possibilities to minimize the total weighted completion time for training multiple models. The execution and communication times are deterministically predicted using a simulation-based performance model.

3.2.1 Input Specification. Formally, we represent the computational graph for each model i as $G_C^i = (V_C^i, E_C^i)$, where V_C^i denotes atomic computational operators (e.g., convolution, matrix multiplication), and E_C^i specifies data dependencies (defining execution order among operators).

The set of available heterogeneous computing devices is denoted by $V_D = d_1, \ldots, d_M$, comprising GPUs, TPUs, and similar hardware units with diverse computational capacities and memory resources. Devices are interconnected via multi-edge physical links, represented as $L_{(d_j, d_k)}$, each with multiple bandwidth capacities $B_{(d_j, d_k)}^\alpha$ reflecting concurrent communication channels with varying bandwidth and latency.

Operator execution time $T_{\text{exec}}(v, d_j)$ for operator v on device d_j is obtained from a simulation model based on device-specific characteristics. The communication duration $T_{\text{comm}}(\text{size}, \ell_\alpha)$ indicates the time required to transfer data of a given size through link ℓ_α.

Memory constraints include operator execution memory, $\text{Mem}op(v)$, representing memory usage during operator execution, and intermediate data memory, $\text{Mem}data(u, v)$, required for storing data transferred from operator u to v.

3.2.2 Output Specification. The solution comprises:

- Device assignment $D(v)$ for each operator v, with execution start s_v and end e_v times.
- Selected communication link $\ell_{(u,v)}$ and the respective start $s_{(u,v)}$ and end $e_{(u,v)}$ times for each data dependency (u, v).

3.2.3 Optimization Objective. The objective is to minimize the weighted sum of the completion times (makespans) for all models:

$$\min_{D,s,\ell} \sum_{i=1}^n \omega_i \cdot T_{\text{compl}}^i, \quad \text{where } T_{\text{compl}}^i = \max_{v \in V_C^i} e_v. \quad (3)$$

Here, ω_i denotes the priority weight for model i, and each model's completion time is determined by its final operator's end time.

3.2.4 Constraints. The primary constraints include:

- **Data Dependency Constraints:** An operator v can begin only after all predecessors and their data transfers are completed:

$$s_v \geq \max e_{u_j}, e_{(u_j,v)}, \quad \forall u_j \in \text{predecessors}(v). \quad (4)$$

- **Communication Constraints:** Data transfer for dependency (u,v) commences only after operator u completes execution:

$$s_{(u,v)} \geq e_u. \quad (5)$$

- **Memory Constraints:** Memory usage on device d_j must not exceed its total capacity M_{d_j}:

$$\sum_{v \in V_{C,j}} \text{Mem}_{op}(v) + \sum_{(v,w) \in E_{C,j}} \text{Mem}_{data}(v,w) \leq M_{d_j}. \quad (6)$$

- **Bandwidth Constraints:** Total bandwidth usage on each link ℓ_α at time t must not exceed its bandwidth limit B_α:

$$\sum_{c \in C(t,\alpha)} \text{rate}(c) \leq B_\alpha. \quad (7)$$

3.3 Algorithm

Algorithm 1 presents our parallel branch-and-bound search method to efficiently explore the optimal operator assignment and scheduling across heterogeneous computing resources.

Initially, the algorithm starts with an initialization procedure, where input data, including computation graphs, device specifications, and resource constraints, are processed. A root node, representing an initial state with all operators unassigned, is created (line 2). The best solution and its upper bound (minimal known cost) are initialized, optionally leveraging a heuristic greedy strategy to quickly provide a baseline solution (line 4). A priority queue is then established to organize exploration nodes by their estimated cost (line 5).

The main parallel exploration procedure (`ParallelSearch`, lines 6–16) proceeds by iteratively examining nodes from the priority queue. At each iteration, the node with the minimal estimated completion cost is selected (line 8). If this node represents a complete assignment (i.e., all operators are assigned), the algorithm compares its cost against the current best-known solution, updating the latter if an improvement is found (lines 9–10).

If the node is incomplete, the algorithm generates feasible child nodes representing possible next assignments of operators, considering current scheduling constraints (lines 12–13). Each feasible child node undergoes a cost estimation procedure (line 13). Nodes whose estimated cost exceeds the current best-known upper bound are discarded early, reducing unnecessary exploration (lines 14–15).

The process repeats until all promising solutions have been explored or pruned, ultimately returning the optimal scheduling solution (line 16).

3.4 Search Space

Our method considers multiple splitting strategies at the operator level, where each operator has a maximum of K possible splits, resulting in a combinational complexity of $O(K^{|V_C^i|})$. At the device level, each resulting sub-operator needs to be mapped onto one of the available devices in $|V_D|$, adding another layer of complexity, up to $O(|V_D|^p)$, where p is the number of sub-operators. Furthermore, scheduling and communication sequences must be

---

Algorithm 1: Parallel Branch-and-Bound Search

Function `Initialize()`:
  Read input graphs, devices, constraints;
  Create root node N_{root} (all operators unassigned);
  best_UB $\leftarrow +\infty$, best_solution $\leftarrow \emptyset$;
  (Optional) Greedy initialization for best_solution, best_UB;
  PQ $\leftarrow$ PriorityQueue(), PQ.$push(N_{\text{root}})$;

Function `ParallelSearch(PQ)`:
  while PQ $\neq \emptyset$ **do**
    $N \leftarrow$ PQ.$pop()$;
    if N is complete solution and $F(N) <$ best_UB **then**
      best_solution $\leftarrow N$, best_UB $\leftarrow F(N)$;
      continue;
    for each feasible child N_{child} of N **do**
      Estimate cost $F(N_{\text{child}})$;
      if $F(N_{child}) <$ best_UB **then**
        PQ.$push(N_{\text{child}})$;
  return best_solution;

---

arranged temporally, which causes the overall search space to grow exponentially.

To cope with this immense search space, we first apply constraints to eliminate infeasible choices. Subsequently, heuristic rules are utilized to effectively reduce the initial search scope, such as presetting initial search points based on multi-edge graph structures and GPU performance. Additionally, techniques such as multithreading can be employed to concurrently simulate and evaluate multiple scenarios, significantly accelerating the search process.

4 Evaluation

We use SimAI to evaluate the effectiveness of our proposed approach. The evaluation considers two scenarios: heterogeneous computation and dynamic network conditions.

Environment Setup. Specifically, we utilize SimAI to simulate task execution times by leveraging the simulated execution results to guide the parameter optimization. SimAI's AIOB invokes CUDA kernel simulations for three distinct real GPUs. Subsequently, the AICB module generates workloads for further simulations and analyses. The GPUs used in this study are as follows:

- **RTX4090D**: Features 14,592 CUDA cores based on the Ada Lovelace architecture, 114 Streaming Multiprocessors (SMs), a boost frequency of 2.52 GHz, 24 GB GDDR6X memory, and fabricated using TSMC's 5nm process.
- **L20**: Equipped with 11,776 CUDA cores based on the Ada Lovelace architecture, 92 SMs, a boost frequency of 2.52 GHz, 48 GB GDDR6 memory, and also utilizes TSMC's 5nm process.
- **V100**: Contains 5,120 CUDA cores based on the Volta architecture, 80 SMs, a boost frequency of 1.38 GHz, 32 GB HBM memory, and is manufactured using TSMC's 12nm process.

Models and Baselines. To demonstrate that our approach effectively identifies efficient plans, we evaluate four representative LLMs: LLaMA_7B, GPT_13B, GPT_22B, and GPT_175B. We compare

(a) RTX4090D and L20 GPUs   (b) RTX4090D and V100 GPUs   (c) Impact of network bandwidth and parallel strategies
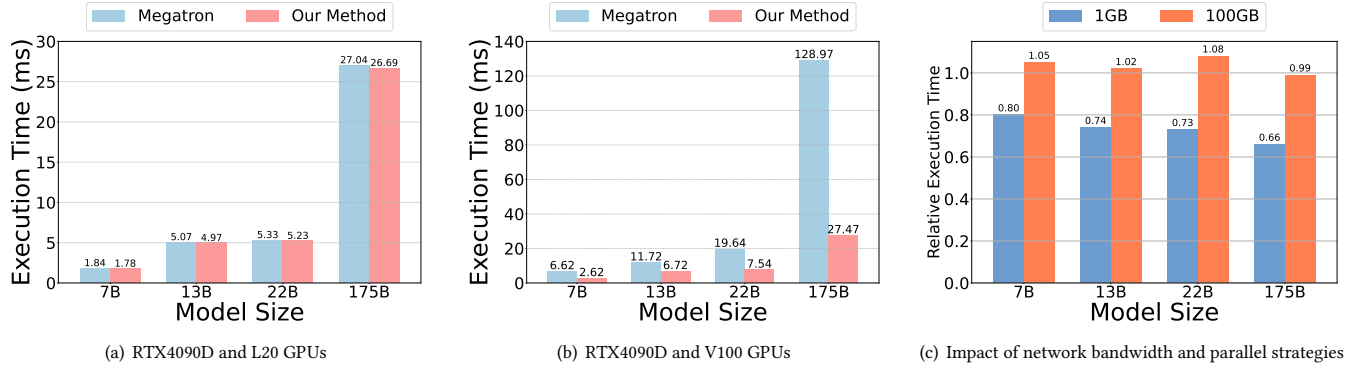
Figure 6: Comparison of execution times for training one epoch under heterogeneous computing devices and dynamic network conditions: (a) RTX4090D combined with L20 GPUs, (b) RTX4090D combined with V100 GPUs, and (c) relative execution times for different tensor parallelism (TP) sizes under varying network bandwidth conditions.

our method with Megatron using its default configuration as a baseline. We consider two evaluation scenarios: heterogeneous computation and dynamic network conditions.

4.1 Scenario 1: Heterogeneous Computation.

We evaluate the performance on clusters comprising 8, 16, 32, and 256 nodes (each containing an equal number of two GPU types) under conditions of both similar and significantly different device performance. Although the RTX4090D and L20 GPUs originate from the same wafer, differences in the enabled functional modules result in minor performance variations. As shown in Figure 6(a), compared with conventional task allocation with equal computational workloads, our approach achieves approximately 1.01 to 1.03 times better performance than the general-purpose Megatron framework by utilizing layer-level task assignment.

For environments with substantial performance disparities, such as integrating the latest RTX4090D GPUs with older V100 GPUs, significant differences in the computation times exist. Consequently, our method substantially enhances performance, achieving speedups ranging from approximately 1.74 to 4.69 times compared with Megatron, as illustrated in Figure 6(b).

4.2 Scenario 2: Dynamic Network Conditions.

We evaluate the execution time for training one epoch across different parallel strategies under varying network conditions using 8, 16, 64, and 256 V100-32G-PCIe GPUs. Figure 6(c) shows the absolute execution time under different network conditions when comparing lower TP sizes to higher TP sizes. Specifically, we set TP sizes of 2 versus 4 for the 7B model, 4 versus 8 for the 13B model, 8 versus 16 for the 22B model, and 16 versus 32 for the 175B model.

We observe that, for models with fewer parameters, a lower network bandwidth increases the execution time by approximately 25% to 52%. This indicates that communication overhead significantly slows down training under low-bandwidth conditions, with higher TP sizes causing even greater increases in the execution time. Conversely, when the network bandwidth is not constrained, larger TP sizes still slightly increase the execution time by approximately 2%

to 8%. However, for very large models, the overhead introduced by higher TP parallelism is offset by the non-overlapping communication time inherent in pipeline parallelism (PP).

5 Limitation

Our study has the following two limitations:

First, due to simulator constraints, we are currently limited to parallel strategies defined by the Megatron-LM. In future work, we plan to implement a native operator-level simulator rather than relying on coarse-grained model-level task assignments.

Second, to accommodate variable-length operators, the search space grows exponentially. Even with multithreading, it is difficult for the CPU to match the search efficiency of existing methods. We plan to leverage FPGA-based acceleration for simulations in future work.

6 Conclusion

This study proposes a multi-edge abstraction to address task execution for LLMs under realistic dynamic network conditions and heterogeneous computing environments. Through experiments, we demonstrate that operator splitting effectively reduces task execution times in heterogeneous settings. Our method significantly outperforms existing frameworks, achieving notable speedups of up to 4.69 times. Moreover, the network conditions significantly influence the search results for parallel strategies. Under varying network environments, newly identified parallel strategies can reduce execution times by up to 52% compared with previously optimal strategies.

Acknowledgments

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[4] NVIDIA Corporation. 2017. NVIDIA DGX-1 System Architecture Whitepaper. https://www.azken.com/images/dgx1_images/dgx1-system-architecture-whitepaper1.pdf. Accessed: 2025-03-10.

[5] NVIDIA Corporation. 2017. NVIDIA DGX-1 with Tesla V100 System Architecture. https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf. Accessed: 2025-03-10.

[6] NVIDIA Corporation. 2020. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf. Accessed: 2025-03-10.

[7] NVIDIA Corporation. 2023. NVIDIA DGX H100 System User Guide. https://docs.nvidia.com/dgx/dgxh100-user-guide/dgxh100-user-guide.pdf. Accessed: 2025-03-10.

[8] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG] https://arxiv.org/abs/2205.14135

[9] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 2021. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 431–445.

[10] Yangyang Feng, Minhui Xie, Zijie Tian, Shuo Wang, Youyou Lu, and Jiwu Shu. 2023. Mobius: Fine tuning large-scale models on commodity gpu servers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 489–501.

[11] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. 2024. ReCycle: Resilient Training of Large DNNs using Pipeline Adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 211–228.

[12] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[13] Muyan Hu, Ashwin Venkatram, Shreyashri Biswas, Balamurugan Marimuthu, Bohan Hou, Gabriele Oliaro, Haojie Wang, Liyan Zheng, Xupeng Miao, Jidong Zhai, and Zhihao Jia. 2024. Optimal Kernel Orchestration for Tensor Programs with Korch. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24)*. ACM, 755–769. doi:10.1145/3620666.3651383

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[15] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. 2023. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 382–395.

[16] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. doi:10.1145/3341301.3359630

[17] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 463–479.

[18] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. 2023. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th annual international symposium on computer architecture*. 1–14.

[19] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems* 5 (2023), 341–353.

[20] Seonho Lee, Amar Phanishayee, and Divya Mahajan. 2025. Forecasting GPU Performance for Deep Learning Training and Inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 493–508. doi:10.1145/3669940.3707265

[21] Dacheng Li, Hongyi Wang, Eric Xing, and Hao Zhang. 2022. Amp: Automatically finding model parallel strategies with heterogeneity awareness. *Advances in Neural Information Processing Systems* 35 (2022), 6630–6639.

[22] Shigang Li and Torsten Hoefler. 2021. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[23] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[24] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[25] Ziming Liu, Shenggan Cheng, Haotian Zhou, and Yang You. 2023. Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[26] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. 2022. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *arXiv preprint arXiv:2211.13878* (2022).

[27] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–15.

[28] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. 2021. The Design Process for Google's Training Chips: TPUv2 and TPUv3. *IEEE Micro* 41, 2 (2021), 56–63. doi:10.1109/MM.2021.3058217

[29] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.

[30] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[31] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[32] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 3505–3506.

[33] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[34] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[35] Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, Mohd Muzzammil, and Myeongjae Jeon. 2024. Metis: Fast Automatic Distributed Training on Heterogeneous {GPUs}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 563–578.

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[37] Xizheng Wang, Qingxu Li, Yichi Xu, Gang Lu, Dan Li, Li Chen, Heyang Zhou, Linkang Zheng, Sen Zhang, Yikai Zhu, et al. [n. d.]. SimAI: Unifying Architecture Design and Performance Tunning for Large-Scale Large Language Model Training with Scalability and Precision. ([n. d.]).

[38] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. doi:10.1145/1498765.1498785

[39] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. 2024. Mirage: A Multi-Level Superoptimizer for Tensor Programs. arXiv:2405.05751 [cs.LG] https://arxiv.org/abs/2405.05751

[40] Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. 2021. Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 503–521. https://www.usenix.org/conference/atc21/presentation/yu

[41] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277* (2023).

[42] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.