Metis: Fast Automatic Distributed Training on Heterogeneous GPUs

Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, and Mohd Muzzammil, *Samsung Research;* Myeongjae Jeon, *UNIST*

https://www.usenix.org/conference/atc24/presentation/um

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Metis: Fast Automatic Distributed Training on Heterogeneous GPUs

Taegeon Um[*,†,§], Byungsoo Oh[*,§], Minyoung Kang[*,§], Woo-Yeon Lee[§]
Goeun Kim[§], Dongseob Kim[§], Youngtaek Kim[§], Mohd Muzzammil[§], Myeongjae Jeon[¶]

[§] *Samsung Research,* [¶] *UNIST*

Abstract

As deep learning model sizes expand and new GPUs are released every year, the need for distributed training on heterogeneous GPUs rises to fully harness under-utilized low-end GPUs and reduce the cost of purchasing expensive high-end GPUs. In this paper, we introduce Metis, a system designed to automatically find efficient parallelism plans for distributed training on heterogeneous GPUs. Metis holistically optimizes several key system components, such as profiler, cost estimator, and planner, which were limited to single GPU types, to now efficiently leverage compute powers and memory capacities of diverse GPU types. This enables Metis to achieve fine-grained distribution of training workloads across heterogeneous GPUs, improving resource efficiency. However, the search space designed for automatic parallelism in this complexity would be prohibitively expensive to navigate.

To address this issue, Metis develops a new search algorithm that efficiently prunes large search spaces and balances loads with heterogeneity-awareness, while preferring data parallelism over tensor parallelism within a pipeline stage to take advantage of its superior computation and communication trade-offs. Our evaluation with three large models (GPT-3, MoE, and Wide-Resnet) on combinations of three types of GPUs demonstrates that Metis finds better parallelism plans than traditional methods with 1.05 ~ 8.43× training speed-up, while requiring less profiling searching time. Compared to the oracle planning that delivers the fastest parallel training, Metis finds near-optimal solutions while reducing profiling and search overheads by orders of magnitude.

1 Introduction

Training large-scale models, such as GPT-3 [4], on extensive datasets necessitates parallelism across a large number of GPUs. In such distributed deep learning (DL) training, it is crucial to choose an appropriate parallelism plan because the strategy of distributing the model and data across GPUs has a significant impact on training efficiency. The choice of parallelism plan directly influences how hardware resources are utilized. Inadequate plans often result in severe communication bottlenecks, highly imbalanced computations, and potential GPU memory overflow. Any of these issues can cause a marked slowdown in training speed and increase the overall cost of training.

Recent advancements in DL systems like Alpa [49] fully automate distributed training [21, 26, 49] to tackle this problem. However, their approaches have limited applicability in many contemporary clusters for DL training, which comprise various GPU generations and different numbers of GPUs per node [16, 44]. Over the last decade, GPUs have undergone consistent improvements, with FLOPs per dollar doubling every two years [12]. As GPU manufacturers continue to develop and roll out new lines of GPUs (e.g., NVIDIA Blackwell [2]), we anticipate a more pressing need for efficient automatic parallelism in clusters with heterogeneity in not only types of GPUs but also the number of GPUs per node.

Concretely, we aim to address two main challenges overlooked by existing systems in practice. First, the search space and methodology of automatic parallelism is currently overly simplistic, frequently leading to sub-optimal plans, and thus needs to be comprehensively expanded. The diverse computing capabilities of different GPUs demand a more fine-grained exploration of model and data partitioning across the GPUs. Second, more attention should be paid to striking a right balance between speed and thoroughness while exploring this complex, newly expanded search space. This ability, which has not been much studied before, is particularly indispensable in shared, non-dedicated clusters where resource allocation is neither predetermined nor static [5, 30]. In such scenarios, any promising approach to automatic parallelism must promptly identify the most effective plan to avoid leaving allocated GPUs idle for a long time.

In this paper, we present Metis, a system designed to automate distributed training on a variety of GPU types. Our key design principle in tackling the first challenge is unraveling in-

* Equal contribution.
† Corresponding author: taegeon.um@samsung.com

trinsic details about these diverse GPUs into core components to jointly optimize automatic parallelism. Metis maintains all possible execution scenarios on heterogeneous GPUs, allowing its profiler to precisely profile necessary performance metrics based on the comprehensive GPU information. Further, Metis is equipped to generate all possible combinations of device groups within a GPU cluster while accommodating both homogeneous and heterogeneous GPU compositions. So, Metis can effortlessly determine whether a device group consists of heterogeneous GPUs or not, which is essential for its cost estimator and parallelism planner to accurately assess model partitioning and parallelism strategies across device groups with various GPU combinations. It is worth noting that previous methods were only able to represent information about a single type of GPUs in device groups that always consist of the same number of GPUs within a group. Such methods cannot be suited for our target scenarios.

As the planner explores the hetero-aware search space larger than the search space on homogeneous GPUs, the second challenge emerges due to the high overheads involved in examining the huge search space and profiling the required metrics. Metis reduces such overheads by harnessing characteristics of DL models and profiling only the required information. Metis estimates the execution time and memory usage of the composition of layers only with profiled metrics for a few layers of a model (e.g., skipping repetitive layers and reusing the metric of a single layer for the repetitive ones). Metis also uses its own memory estimation model to measure the memory usage for the composition of layers with fewer profiling overheads. In exploring search space, Metis develops a new algorithm that efficiently prunes large search spaces by filtering out similar combinations of device groups while preferring data parallelism over tensor parallelism within a stage and balancing data and layers with capacity-aware load balancing across heterogeneous GPUs.

跳过重复的 layers

相较于 TP 更偏好 DP，pipeline load balance

We have implemented Metis on top of Alpa [49] with $3K+$ lines of codes. We evaluate Metis with three DL models by varying their sizes on diverse sets of heterogeneous GPUs (V100, P100, and T4 on GCP). Our evaluations show that Metis improves the training performance by $1.05 \sim 8.43\times$ with less profiling and searching overheads compared to the state-of-the-art work (AMP and Alpa). Moreover, Metis finds near-optimal solutions while significantly reducing profiling and searching overheads compared to the oracle search and full profiling.

We make the following contributions in this paper:

- We investigate existing work for automating distributed DL training and find that it is not optimized for heterogeneous GPU environments.
- We design and implement Metis, that automatically finds efficient parallelism plans on heterogeneous GPUs, by optimizing the profiler, cost estimator, and planner altogether.
- We develop a new efficient search algorithm to find near-optimal plans and a cost model with a significant reduction of searching and profiling overheads.
- We thoroughly evaluate Metis with diverse sets of heterogeneous GPUs and various models and show the effectiveness of Metis.

2 Distributed Deep Learning Training

Training a deep learning (DL) model involves processing a minibatch of training data through a sequence of layers in order during forward pass (FP) and then in reverse order during backward pass (BP). At the core of this bidirectional FP and BP is calculating gradients for the model parameters, which are used to update the model. This process is repeated until the model reaches convergence.

In distributed training, the layers of the model and their associated input data are partitioned across multiple GPUs. This mainly serves two purposes: (1) to alleviate memory pressure on each GPU, and (2) to accelerate FP-BP operations by processing them across GPUs in parallel. The performance of distributed training can exhibit significant variations based on strategies taken for parallelism, including parallelism methods, device grouping, and load balancing. These strategies represent crucial dimensions in the search space, with specific configurations defining a *parallelism plan* for model training. This section gives a comprehensive overview of these aspects.

2.1 Parallelism Methods

Our work focuses on three popular forms of parallelism – pipeline parallelism (PP), data parallelism (DP) and tensor parallelism (TP) – although other methods like sequence-parallelism [23] or ZeRO [34,35,42] do not restrict optimizing automatic parallelism. Each of these methods allows for a distinct trade-off between computation, communication, and memory costs, as follows:

• ***Pipeline parallelism.*** PP divides the layers of a model into multiple *stages*, each comprising consecutive layers, and assigns these stages to groups of GPUs, known as *device groups*. Thus, during model training in PP, data communication is predominantly limited to adjacent layers spanning different stages. Current PP methods optimize GPU utilization by feeding data into stages on different device groups in a pipelined manner [13, 28, 29]. These methods typically split a minibatch into smaller micro-batches and orchestrate their execution through the pipeline.

PP proves to be effective for training large DL models with memory demands that far exceed the capacity of a single GPU. Nonetheless, the way PP execution is structured introduces a notable trade-off between memory and communication costs. In general, increasing the degree of PP elevates system-wide

communication because it operates more stages, but it simultaneously reduces memory pressure on each device group hosting a smaller individual stage. Note that, at times, the entire model forms a single stage that occupies a single device group comprising all GPUs.

• ***Data parallelism.*** Several parallelism methods exist to exercise GPUs in a device group, with DP being one of them. DP allows each GPU to train the same layers in a stage using a disjoint subset of input data [8,51]. As the training proceeds with local data on different GPUs within the DP group in parallel, the gradients obtained upon finishing FP and BP should be all-reduced to synchronize model parameters.

For DP, there is a trade-off between computation and communication costs. The communication cost increases as the number of GPUs in the DP group grows. But, this makes the time required for each GPU to complete FP and BP decrease, thanks to the efficiency gains from processing data with a higher degree of data parallelism.

• ***Tensor parallelism.*** TP is another complementary way to effectively utilize GPUs in a device group. Unlike the data-level partitioning seen in DP, TP partitions the layer weights and computations of a stage into non-overlapping smaller chunks called *tensors* [31, 38]. These tensors are then distributed across multiple GPUs that constitute the TP group. Each GPU handles a specific tensor for every split layer and participates in synchronous communication with other GPUs in the group. The goal is to merge outputs while collaboratively training each split layer in the stage, ensuring consistency throughout the FP and BP. Overall, a higher TP degree alleviates memory pressure and computation time on each individual GPU, but it also accompanies higher communication overhead among the GPUs in the TP group.

2.2 Device Grouping

In a broader context, a device group can vary in size and consist of either homogeneous or heterogeneous GPUs. For instance, consider a scenario with two GPU nodes: one with 2×V GPUs and the other with 2×T GPUs, where V and T represent GPUs with different computing powers. Assuming the number of GPUs is in powers of 2, this cluster allows for the formation of six distinct device groups: (V), (T), (V, V), (T, T), (V, T), and (V, V, T, T), where a PP stage can be executed on any of these device groups.

Due to the diverse GPU compute capabilities, execution times for a stage configured with specific DP and TP settings can exhibit significant differences across device groups, even when they are of equal size. Furthermore, the range of potential device groups will expand in practical scenarios where nodes host widely varying numbers of GPUs or when the cluster includes many different GPU types. Current systems do not fully account for such diversity in device groups. We delve deeper into this topic in § 3.

2.3 Load Balancing

Traditional DL frameworks like Megatron-LM [38] and DeepSpeed [36] support *uniform* partitioning of data and layers, which is tailored for homogeneous GPUs. However, *non-uniform* partitioning becomes more valuable when it comes to heterogeneous GPUs as they differ in computing and memory capacities. This ability enables the workload to be distributed across GPUs in a fine-grained manner, thereby minimizing GPU idle times.

Load balancing that steers this non-uniform partitioning broadly falls into two categories: *layer load balancing*, which manages the layer distribution across PP stages, and *data load balancing*, which adjusts local batch sizes across GPUs in a DP group. Consider the latest NVIDIA H100 GPU [3]; its superior computing power makes it more apt for handling a larger quantity of layers or data than older models like the V100 GPU [1] during stage execution. Importantly, this load balancing needs to factor in not only the computational power but also the memory capacity of each GPU type, given diverse memory specifications – e.g., 80 GBs for H100 *vs.* 16 or 32 GBs for V100. Therefore, a desirable load balancing strategy should faithfully accommodate actual resource usage patterns across all device groups and GPU types.

3 Current Limitations for Auto-Parallelism

Several DL systems, including Alpa [49], AMP [21], and Galvatron [26], have been developed to simplify the complex task of manually finding efficient distributed training plans. Their key technique is automating this process through the analytical comparison of a large number of training plans, using cost models and profiled metrics that enable a timely estimation of each plan's execution time. To achieve this, these systems must effectively explore a huge search space encompassing parallelism, device grouping, and load balancing strategies.

The search algorithms proposed so far normally operate at two nested levels. At the higher level, they evaluate all possible combinations of PP stages and device groups, which are derived from how the model's layers and available GPUs are partitioned. Once a stage is assigned to a device group, the search algorithm shifts to the next level, where it seeks the most effective mix of intra-stage parallelism. This involves varying DP and TP degrees within the stage, while taking into account the constraints posed by the device group (e.g., number of GPUs, GPU memory capacities).

Despite their effectiveness in simpler hardware settings, current search algorithms are prone to miss highly preferable parallelism plans in GPUs with non-identical specifications. Due to their homogeneous GPU assumption in their system design, key performance metrics (e.g., execution time and memory usage for each layer) of heterogeneous GPUs are profiled but abstracted as homogeneous ones. For instance, Alpa [49] and AMP [21] simply take an average of the com-

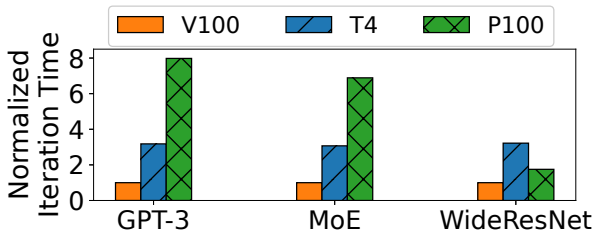

Figure 1: The normalized execution time of a minibatch of models on various GPUs.

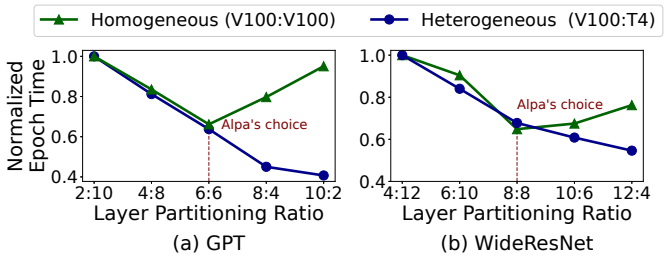

Figure 2: The normalized epoch time in various layer load balancing on two GPUs (homogeneous: two V100 GPUs, heterogeneous: one V100 and one T4). Regardless of homogeneous or heterogeneous GPUs, Alpa chooses the same layer load balancing strategy due to its homogeneous GPU abstraction (uniform layer balancing across GPUs).

puting power across GPUs and operate on the premise that all GPUs have this averaged compute capacity [*].

This approach, while simplifying the system design, hinders systems to explore parallelism plans with diverse device groups and load balancing strategies and introduces the following limitations:

1) Limited search space in device groups. Previous search algorithms often encounter challenges in effectively exploring device groups, frequently resulting in the generation of sub-optimal parallelism plans. This issue arises mainly because the optimal plan for nodes equipped with heterogeneous GPUs may not be within the search space designed to explore. For instance, in the scenario described in § 2.2 with two GPU nodes, conventional methods consider only three device groups (as opposed to six originally appeared in § 2.2): (R), (R, R), (R, R, R, R), where R denotes an averaged abstract GPU representing both V and T. If V and T have dramatically different computing powers, the overall performance is likely to vary significantly depending on which stage is assigned to which GPU type. To illustrate, we measure the average time required to process a single minibatch for three different DL models, using the largest minibatch size that fits into the GPU memory of single-GPU device groups. The results in Fig. 1 show that all models exhibit considerable variability in execution times across the three types of GPUs we examined. Nonetheless, previous methods do not account for such alternate device placements and ordering in the exploration scope due to their dependence on the abstracted GPU model R.

Another major problem is that most automatic parallelism approaches are tailored to support sets of GPUs configured in a two-dimensional $M \times N$ grid, where M is the number of GPU nodes and N is the number of GPUs per node. This design inherently limits their adaptability to GPU sets that deviate from this two-dimensional structure, such as those with an uneven number of GPUs per node. Such heterogeneity is quite common in modern GPU clusters where a fraction of GPUs within a node are being used, with a preference for high-end GPUs than low-end GPUs, which results in different numbers of available GPUs per node and GPU type. Addressing all these limitations is crucial for automatic parallelism to be prevalently applied to the diverse GPU environments of today.

2) Limited load balancing. Treating all GPU resources as homogeneous also makes it difficult to properly balance computational loads across GPUs. This issue is highlighted in Fig. 2, which presents a simple layer partitioning scenario in PP. Given V100 and T4 GPUs, an optimal strategy would be to allocate more layers to the V100, considering its superior computing power compared to the T4. Existing load balancing approaches, however, cannot accomplish this level of specific layer allocation because they are unable to differentiate *qualitatively* between these two types of GPUs. Moving forward, there is a need to automatically find the most efficient load balancing strategy while expanding the exploration space to include all parallelism methods, i.e., PP, DP, and TP.

3) Time-consuming search algorithm. A larger search space increases the chances of discovering better parallelism plans, yet it also requires longer exploration times. Given the complexities introduced by resource heterogeneity, which broadens the scope of device groups and load balancing factors, we put our emphasis on spending short time in navigating a search space of a large size. Such a strategy makes the search tractable, especially in shared GPU clusters where resources are dynamically allocated from a server pool and require timely utilization for high cluster efficiency. To realize this goal, the focus should be on including potential plans that critically impact performance, while omitting those with little to no impact.

Identify potentially optimal plans and omit those with little to no impact

Table 1 provides a comparative analysis of search space parameters for two popular systems, AMP [21] and Alpa [49], in the first and the second columns, respectively. Key challenges in current search strategies are to explore extensive combinations of stage-device group pairs, intra-stage parallelism (including data load balancing) within a stage, and layer load balancing across stages. For instance, all possible combinations of assigning stages to the device groups are ${}_S\Pi_G$ (S: the number of stages, G: the number of possible device groups),

[*]Alpa can average the computing power of heterogeneous GPUs by turning off the `use_hlo_cost_model` option [41] with high profiling overheads.

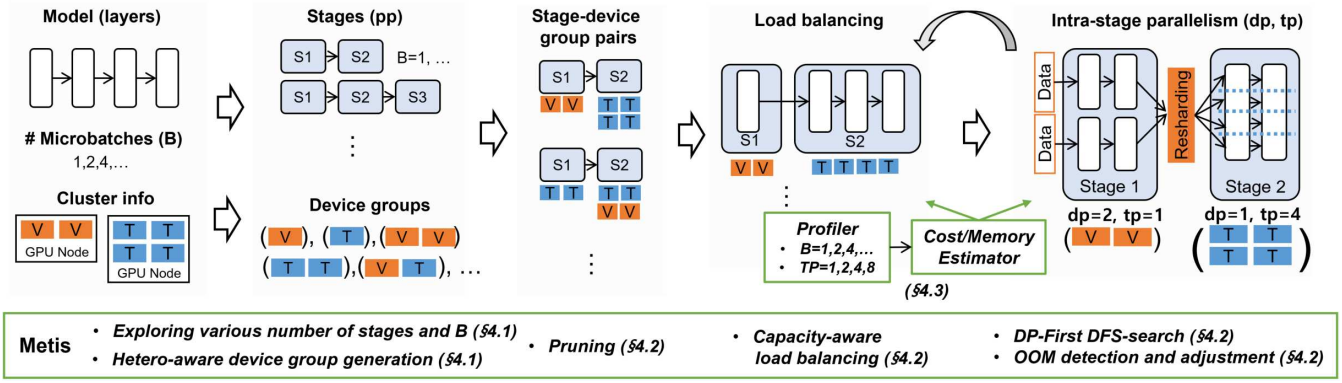

Figure 3: An overview of how Metis automates parallelism plans on heterogeneous GPUs.

which extremely increases when the number of heterogeneous GPUs and layers is large. Moreover, each stage can have diverse plans of intra-stage parallelisms and layer load balancing. As an example, Alpa examines detailed parallelism plans for each stage using an integer linear programming (ILP) solver that optimizes fine-grained operator-level partitioning [*]. This approach creates a substantial search space and considerable search overheads. In addition, a common technique for layer partitioning evaluates all layer distribution possibilities across stages through dynamic programming [21, 49], with time complexity reaching around $O(L^3)$, where L is the layer count. Based on our empirical findings (§ 5.4), search and profiling times can easily exceed tens of hours for a cluster comprising just 16 GPUs.

4 Metis Design

Metis is designed to address the above limitations to find better parallelism plans for distributed training on heterogeneous GPUs. Specifically, Metis is aware of the characteristics and the performance behavior of DL training on heterogeneous GPUs to optimize various components such as the search space, searching algorithm, and profiler with cost model.

In this section, we first illustrate how Metis expands the existing search space to find better plans (§ 4.1). Next, we present a Metis's efficient search algorithm that reduces huge search overheads on the expanded search space (§ 4.2) and also show an efficient profiling and cost estimation (§ 4.3). The overview of Metis design is illustrated in Fig. 3.

4.1 Hetero-Aware Search Space

Metis strategically prevents the overall search space from exploding by deliberately simplifying the exploration of intra-stage parallelism, while expanding the search scope for device groups and load balancing. This decision is reasonable considering the performance impact of heterogeneous devices with different device groups and load balancing strategies as shown in § 2.2. For intra-stage parallelism, Metis takes the middle-ground between AMP and Alpa, allowing different intra-stage parallelism across stages like Alpa but exploring the possible degrees of DP and TP for each stage, with their product (i.e., the two degrees multiplied) compatible with the GPU count of the device group running the stage.

Expanding device groups. Metis can create all possible combinations of device groups in heterogeneous GPUs by selecting k number of GPUs for each node ($0 \le k \le K$, K is the number of available GPUs of each node). Picking several GPUs from each GPU node to compose a device group may generate non-unique combinations of device groups that have

Params/System	AMP	Alpa	Metis
# of stages (pp)	$O(\sqrt{D})$	O(L)	O(L)
Device group	O(1)	O(N+log(M))	O(N* (N+log(M)))
Intra-stage parallelism (dp, tp)	$O(\sqrt{D})$	O(L*N*M *(N+log(M)))	O(D/L)
Load balancing (data)	No	No	$O(\sqrt{gbs})$
Load balancing (layer)	$O(L^3)$	$O(L^3)$	$O(L^3)$
Number of micro-batches	$O(\sqrt{gbs})$	Manual	$O(\sqrt{gbs})$

Table 1: Comparison of *full* search space for auto-parallelism. L is the number of layers, gbs is the global batch size, D is the total number of GPUs, N is the number of nodes, and M is the number of GPUs per node. The existing system supposes $D = N \times M$, whereas Metis supports heterogeneous GPU clusters with different numbers of available GPUs per node. In Alpa, L is determined by grouping operators, and L is also configurable in Alpa. Metis develops an efficient search algorithm (§ 4.2) to avoid the full search of the huge search space.

[*]Alpa groups a set of operators as a layer.

the same computing, memory, and network configuration.

To generate unique combinations of device groups that can differ in the execution time of a stage, Metis compares the following three factors: 1) the number of GPUs (the size of a device group), 2) the types of GPUs, and 3) the node of GPUs in the device groups. The number of GPUs and their types as well as the node of GPUs can distinguish the different computing and memory capacity and network configuration. First, two device groups are inherently different if they have different numbers of GPUs because each device group will have different computing and memory capacity. When the size of device groups is equal, Metis compares the number of each type of GPU between device groups. If these are also the same, Metis then finally compares the node of each type of GPU to distinguish network configurations. According to the above rules, various device groups are generated. For instance, even if there is one V GPU in a node and three T GPUs in another node that breaks the two-dimensional cluster assumption of the existing work, Metis can compose a device group with (V, T, T, T).

Load balancing. For layer load balancing, Metis reduces the search time by exploiting the characteristics of DL models and modeling the execution time in our cost model, which is further illustrated in § 4.2. For data load balancing, Metis differentiates the size of micro-batches across heterogeneous GPUs. For instance, when the micro-batch size is bs, and there are N heterogeneous GPUs within a device group, Metis explores diverse sizes of micro-batches: $\sum_{i=1}^{N} mbs_i = bs$ where mbs_i is the size of a micro-batch of i-th GPU ($1 \leq mbs_i$). The i-th GPU then processes mbs_i number of input data during forward and backward computations, which requires different amounts of computations with different sizes of mbs.

Number of micro-batches. Metis also explores different numbers of micro-batches because a micro-batch size * is an important factor for GPU utilization and amount of memory used in the GPU. As modifying a global batch size may affect convergence, we keep the global batch size while exploring the micro-batch sizes. Large micro-batch size can improve the GPU utilization and FLOPs, but it may cause an out-of-memory (OOM) exception in a GPU. Such OOM exceptions can be exacerbated in heterogeneous GPUs where each GPU has different memory capacity.

4.2 Hetero-Aware Cost and Search

Although Metis judiciously defines the hetero-aware search space by reducing the size of intra-stage parallelism and considering the expanded device groups and load balancing, the number of possible candidate plans to explore is still large, which prevents a fast decision of automatic distributed training. This quick decision is important in dynamic resource environments where resource configurations may dynamically change for efficient cluster resource scheduling [22], or crucial for cluster-level schedulers to find resource-efficient GPU configurations before running a job on heterogeneous GPUs [15, 45].

*A micro-batch size is calculated by dividing a mini-batch size by the number of micro-batches.

4.2.1 Cost Model

To compare costs across plans, Metis finds a solution that minimizes the following cost (T^*) based on the 1F1B scheduler [28] and existing work [21, 49]:

$$T^* = \min\{\sum_{i=1}^{S} t_i + (B-1) \cdot \max_{1 \leq j \leq S} t_j + DP_{all}\} \quad (1)$$

where S is the number of stages, B is the number of micro-batches, and DP_{all} is the communication overhead of DP all-reduce. t_i is the latency of executing i-th stage, and the time includes both forward and backward computations as well as communication overheads across stages (for PP) and layers (for TP).

Metis calculates the PP and DP communication cost using a cost model that considers network bandwidth heterogeneity. The PP cost is derived by dividing the activation size by the inter-stage network bandwidth, while the DP cost is determined by dividing the total size of the model synchronization parameters by the slowest network bandwidth within the DP group. In contrast, Metis profiles the TP communication cost because the frequent and fine-grained TP communications—occurring between operators compared to PP's inter-stage and DP's post-batch timing—might lead to less precise cost estimations. Since TP typically occurs within a single node to mitigate high communication costs, heterogeneities in network bandwidth between nodes do not impact TP communication costs within a node.

All stages fully use the entire GPUs in the heterogeneous GPU cluster, so summing up all GPUs of executing stages will lead to the same shape of the GPU cluster.

4.2.2 Search Algorithm

Algorithm 1 illustrates Metis's hetero-aware search algorithm to minimize T^* in Equation 1. The search algorithm consists of three parts: 1) pruning combinations of stage-device mapping, 2) navigating efficient intra-stage plans with OOM detection based on the trade-off of computing and memory between DP and TP, and 3) balancing layers across stages and data within the stage with capacity-aware allocation. Once possible plans are generated from the search, Metis calculates the estimated time (cost) of each plan and finds the best plan that leads to the minimum estimated cost (Lines 16—19).

1) Pruning combinations of stage-device group mapping. Metis iterates all possible number of stages as well as the number of micro-batches B, but due to a huge search space (§ 3), Metis filters out combinations of stage-device group

Algorithm 1: Hetero-Aware Search Algorithm

```
1  Input: GBS: Global batch size, CE: CostEstimator
2  Output: The best plan (BestPlan) that minimizes T*
3  T* ← ∞, BestPlan ← None
4  for B, Stages, DeviceGroups in EnumerateWithPruning do
5      IntraPlan = {}, MbsPlan = {}, LayerPlan = {}
       /* Initialize intra-stage plans                    */
6      for i from 0 to len(Stages) - 1 do
7          S= Stages[i], Dg = DeviceGroups[i]
8          DP = Dg.GetSize()
9          IntraPlan[S] = (DP, 1)
10         MbsPlan[S] = [GBS / B / DP] * DP
11         if Dg.IsHetero() then
12             MbsPlan[S] = DataLB(S, MbsPlan)
13     StagePlan = [IntraPlan, MbsPlan, LayerPlan]
14     LayerLB(B, Stages, StagePlan)
       /* DFS Search                                      */
15     Plans = (DFS(0, StagePlan, []))
16     for Plan ∈ Plans do
           /* Check valid plans                           */
17         if len(Plan) == len(Stages) then
18             if T* > CE(Plan) then
19                 T* ←CE(Plan), BestPlan ← Plan
20 Function DFS(i, StagePlan, Plan)
21     Plans = [], S ← Stages[i]
22     if i >= len(Stages) or InvalidIntraPlan(i, StagePlan) then
23         Return [Plan]
24     IntraPlan, MbsPlan, LayerPlan ← StagePlan
25     if DetectOOM(S, StagePlan) then
26         CheckpointCurrPlans()
           /* Increase TP                                 */
27         IntraPlan[S] ← (IntraPlan[S][0]-1, IntraPlan[S][1]+1)
28         MbsPlan[S] ← DataLB(S, MbsPlan)
29         LayerPlan[S] ← LayerLB(S, LayerPlan)
30         Plan[i] = [(IntraPlan[S], MbsPlan[S], LayerPlan[S])]
31         Plans.Extend(DFS(i, StagePlan, Plan))
           /* Restore and adjust layers                   */
32         RestoreCheckpointedPlans()
33         LayerPlan[S] ← LayerLB(S, LayerPlan)
34         if DetectOOM(S, StagePlan) then
35             Return []
36     P = [(IntraPlan[S], MbsPlan[S], LayerPlan[S])]
37     Plans.Extend(DFS(i+1, Plan + P))
38     Return Plans
```

pairs based on the following key observations with constraints. Such constraints are knobs that reduce the overhead of searching, when the number of stages and GPUs is large.

First, the variance of size of each stage is not significantly large in optimal solutions. For instance, it is not practical to compose two stages with one GPU and 128 GPUs, respectively (high variance of stage sizes) for optimal performance. Therefore, Metis limits the exploration of device groups with a lower bound of the size of device groups. To vary the lower bound in terms of number of stages, Metis explores device groups with sizes only larger than $\frac{N_{GPUs}}{S} * (1 - var)$ where var is a parameter to limit the exploration of high variant size of device groups.

Second, similar combinations of stage-device pairs lead to similar performance. As an example, when assigning 5 stages ($s_1 \rightarrow s_2$, ..., $\rightarrow s_5$) into 5 device groups (g_1, ..., g_5) where g_1, ..., g_4 have one same GPU and g_5 has two GPUs, we should permute all possible combinations of stage-device group pairs, which results in $binom(5,1) = 5$ combinations*. However, assigning g_5 to s_2 or s_3 (across neighbor stages) will not significantly change the performance if the number of stage is large or model layers are repeated. To reduce search overheads, Metis then filters out such combinations by grouping two device groups with smallest size into a pair and iterating this process until the permutation length is less or equal to the maximum permutation length. For instance, when the permutation length of device group is reduced from 5 to 3, Metis groups (g_1, g_2) into g'_1, and (g_3, g_4) into g'_2, creating G' =[g'_1, g'_2, g_5]. Metis then permutes the elements of G', which results in $binom(3,1) = 3$ combinations, and assigns stages to the device groups in order for each combination.

2) Navigating intra-stage plans with DFS-order. For each combination, Metis efficiently navigates intra-stage parallelisms with an observation that increasing DP (data parallelism) is better than TP (tensor parallelism) within a stage in terms of reducing the execution time [38, 49] (Lines 20—39). This is because the communication overhead of TP is usually larger than that of DP due to the frequent synchronization of activations. Therefore, Metis first initializes all intra-stage plans by maximizing the DP degree (Lines 8—9). However, DP increases the memory pressure of a GPU more than TP, which may cause an OOM exception. Therefore, Metis must consider both the execution time and memory pressure to search near-optimal plans.

Keeping this characteristic in mind, Metis develops a *DP-first* DFS search algorithm that explores the data-parallelism option first and searches other options in the DFS order when there is a possibility of OOM exceptions. Optimistically, if there is no OOM, Metis just performs DP within a stage without exploring other plans. However, if the current stage is estimated to raise an OOM exception based on profiled metrics, Metis explores other plans to mitigate the OOM (Lines 25—35). Metis estimates the memory usage of each layer based on the profiled metrics as described in § 4.3 and calculates the stage memory by summing up the estimated memory of layers within the stage. When the estimated stage memory exceeds its device memory capacity, Metis detects the stage as OOM.

To reduce the memory pressure of the OOM stage, Metis

*Selecting a pair of (s_i, g_5) and assigning the remaining stages into the remaining same device groups.

explores two pathes. The first one is to adjust the intra-stage plan, by increasing the TP degree while decreasing the DP degree (Lines 27—31). The second one is to rebalance layers while retaining the current intra-stage plan (Lines 32—35). Metis distributes the layers of the OOM stage (s_i) across remaining stages proportionally to the computing capability until OOM does not happen in s_i.

3) Capacity-aware load balancing. Metis balances layers across stages based on computing capacity-aware balancing (Line 10 and 29). For load balancing, Metis estimates 1) the capacity of each stage running on a device group and 2) the load of layers. For 1), Metis estimates the throughput of executing a model according to the intra-stage plan of its assigned stage, assuming there is a single stage for executing a model (e.g., S=1 in Equation (1)). For 2), Metis estimates the execution time of each layer. Based on 1) and 2), Metis balances the load of layers across stages proportional to the capacity of each stage.

This computing capability-based allocation reduces the search cost of load balancing and finds efficient allocation strategies without exhaustive search like dynamic programming. During the DFS search, when the intra-stage plan is adjusted, the capacity has been changed according to the intra-stage plan. Metis therefore rebalances layers according to the adjusted capacity of stages to maximize throughput (Line 29) in the adjusted intra-stage plan. Estimating the throughput of a device group according to the intra-stage plan is based on the execution time of layer profiling, which is described in the following section.

4.3 Cost Estimation with Profiling

To run the search algorithm, Metis must estimate the execution time and cost (latency) of layers on a stage, the communication overhead, and the peak GPU memory usage on the stage to detect OOM exceptions.

There is a trade-off between metric profiling and cost estimation in terms of accuracy and profiling overheads. The more we profile the actual metrics, the more the estimated cost is accurate. However, it is infeasible to profile all required metrics for the cost and memory estimation. Naive profiling requires executing all possible candidates of layer execution on various stage-device pairs. Therefore, building a cost model to estimate the execution time and the peak memory usage with a few profiled metrics is required.

For reducing the profiling overhead of execution time (and throughput), Metis harnesses two observations: one is that the execution time of layers can be estimated by summing up the execution time of individual layers with a negligible error, and the other one is that the execution time of repetitive layers can be estimated with profiling a single layer. Therefore, Metis profiles the execution time of individual layers for each GPU type with various tp degree and micro-batch sizes (mbs), while skipping repetitive layers:

$$Time(L^{tp}_{mbs}(i,j)_{gpuA}) = \sum_{i}^{j} Time(L^{tp}_{mbs}(i)_{gpuA})$$

where $L(i,j)$ is the composition layers from i-th to j-th layers.

In contrast with layer execution time, the peak memory usage of layers cannot be accurately estimated by summing up the memory usage of individual layers, because the runtime engine may reuse input/output buffers during the execution of layers with shared variables [14]. Therefore, it is required to profile the memory usage of the composition of layers to estimate peak memory use. However, profiling all compositions of layers for each stage requires $O(L^2)$ number of profiling, which leads to high profiling overheads. Metis reduces the memory profiling overheads with the following heuristics to bind the time complexity to $O(k \cdot L)$, where k is the maximum number of composition layers. First, Metis profiles the memory usage for one GPU type and reuses them for different types of GPUs, since the required memory of executing layers is the same regardless of GPU types. Second, Metis profiles the memory usage of the following layers:

$$Mem(L^{tp}_{mbs}(i,j)) \quad \forall 0 \le i \le L-k \quad \text{and} \quad i \le j \le i+k$$

With the measured memory, Metis can estimate the memory usage of layers as follows [*]:

$$Mem(L(i,j)) = Mem(L(i,j-1)) + Mem(L(j))$$

$$Mem(L(m)) = \sum_{n=m-k}^{n=m-1} \frac{(Mem(L(n,m)) - Mem(L(n,m-1)))}{k}$$

In addition to reducing the memory profiling, Metis profiles representative numbers of tp and mbs, such as $tp = 1,2,4,8$ and $mbs = 1,2,4,8,16...$ When non-profiled tp or mbs (e.g., $mbs = 7$) is required to estimate the cost and memory while searching plans in Algorithm 1, Metis estimates the metrics with the combination of the profiled tp and mbs. As an example, for $mbs = 7$, Metis sums the profiled results of $mbs = 1$, $mbs = 2$, and $mbs = 4$.

Model / # of GPUs	4	8	16	32
GPT-3	1.3B	2.6B	6.7B	15B
MoE	1.3B	2.4B	10B	27B
Wide-ResNet	1B	2B	4B	6.8B

Table 2: The size of model parameters used in our evaluation.

5 Evaluation

We answer the following questions in this evaluation:

[*]We omit tp and mbs for simplicity.

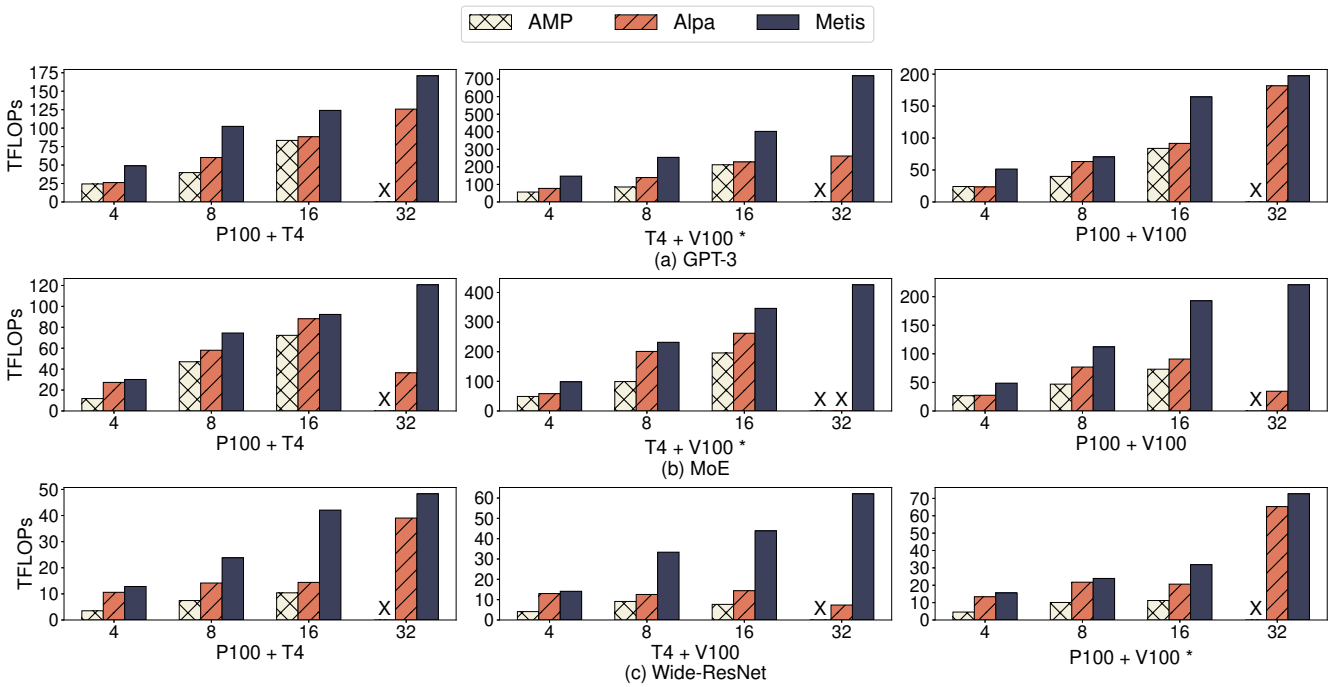

Figure 4: Performance results on diverse environments with three different types of GPUs. *x*-axis is the total number of GPUs with Table 3 GPU configuration. Asterisk mark (*) illustrates the best GPU combination for each model. OOM errors are denoted with X marks.

- When and why does Metis improve performance compared to the state-of-the-art (SOTA) work in various GPU environments and diverse models (§ 5.2)?
- What is the effectiveness of Metis' load balancing and searching algorithm compared to the SoTA (§ 5.3)?
- How much does Metis reduce profiling and search overheads compared to the oracle and SOTA (§ 5.4)?

5.1 Environment and Setup

We have implemented Metis on top of Alpa [49]. We use the Alpa runtime to profile and execute distributed training because Alpa runtime supports different intra-stage parallelisms between stages with cross-mesh resharding. We have implemented Metis's cost model, profiler, and searching algorithm on Python 3.9 with around 3,000+ lines of code. Although we execute the plan on the Alpa runtime, Metis's planner is agnostic to the runtime and can be easily integrated with other runtimes if the runtime supports different intra-stage parallelisms across stages.

Heterogeneous GPU environment. We conduct experiments on the Google Cloud Platform (GCP) and use three types of GPUs: NVIDIA P100 16GB, T4 16GB, and V100 16GB to emulate heterogeneous GPU clusters and environments. We vary the number of GPUs per node for each type of GPU to evaluate Metis on diverse heterogeneous GPU environments. In the main evaluation, we vary the number of GPUs per node from 2 to 8. For each node, we set the CPU and the main memory specification to avoid the CPU or memory bottleneck. In detail, we allocate x CPU cores ($x = 4 * \text{number\_of\_gpus}$) and use n1-standard machines where the machine memory is proportional to the number of GPUs (memory size = 7.5 GB * number of CPU cores). The interconnect network bandwidth between nodes is 16 Gbps.

Models and baselines. To show that Metis finds efficient plans regardless of model architectures, we use three large-scale models: GPT-3 [4], GShard Mixture-of-Experts (MoE) [20], and Wide-ResNet [47] because each model has different model architectures. We will illustrate how Metis decides plans according to the different models. The configuration of the models used in our evaluation is described in Table 2. As baselines, we use two state-of-the-art systems for automating distributed training, AMP [21] and Alpa [49].

5.2 Performance Comparison

This section shows the performance improvement of Metis compared to AMP and Alpa. To fairly compare parallelism plans without the difference between execution runtime, we port the best plan generated from AMP to be executed on the Alpa runtime.

To show the effectiveness of Metis in diverse heterogeneous GPU environments, we compose two types of GPUs:

Number of GPUs	4	8	16	32
GPU A	1x2	1x4	2x4	4x4
GPU B	1x2	1x4	2x4	4x4

Table 3: The GPU setup of Fig. 4. *N*x*M* represents *N* nodes with *M* GPUs for each node. For instance, with 4 numbers of total GPUs in Fig. 4, each type of GPU has one node with two GPUs (1x2).

P100 and T4, T4 and V100, and P100 and V100, increasing the number of each type of GPU from 2 to 8. We have also evaluated systems on environments consisting of three types of GPUs (V100, T4, and P100) but omitted the results as we observed similar behavior to using two types of GPUs.

Here, we set up various environments: uniform GPU composition where different nodes have the same number of GPUs (§ 5.2.1), and non-uniform GPU composition where different nodes have different numbers of available GPUs (§ 5.2.2). In addition to the heterogeneous GPUs, we also evaluate systems on homogeneous GPUs and observe that Metis achieves comparable performance compared to the existing work, which represents that Metis also covers auto-parallelism on homogeneous GPUs.

5.2.1 Uniform GPU composition

Fig. 4 shows the performance of each system in the uniform number of GPUs per node with three models and with various combinations of GPUs. The configuration of GPUs is explained in Table 3.

Across all experiments, Metis outperforms AMP and Alpa on average 2.65× and 2.02×, respectively, finding more efficient parallelism plans on diverse resource environments. Overall, Alpa has better performance than AMP because Alpa explores a more broader search space than AMP as illustrated in Table 1. The performance behavior shows different patterns with respect to different combinations of GPUs and model architectures. We provide a detailed analysis of the performance in the following sections.

Different combinations of GPUs. Regardless of the systems, the performance of distributed training for each model is the highest in the GPU combinations that have the maximum aggregate computing capacity based on Fig. 1. As an example, the performance of T4+V100 is the highest in all cases for GPT-3 and MoE because the combination of T4 and V100 GPUs leads to the maximum aggregate compute capacity for executing the models, whereas WideResNet has the best performance on P100+V100 combination of GPUs. GPT-3 and MoE have the similar performance pattern as they have similar model architectures with a slight difference: they consist of repeated Transformer layers, but MoE uses sparsely activated feed forward networks called experts instead of the dense ones to conditionally leverage a subset of parameters [37]. In GPT-3, the performance on T4+V100 is 3.23× and 3.14× higher than the performance on P100+T4 and P100+V100, respectively. This performance enhancement is proportional to the aggregated computing power of GPUs.

The performance improvement of Metis compared to the existing work also changes according to the different combinations of GPUs. Specifically, as the compute power gap between different types of GPUs increases, Metis shows a larger speedup compared to Alpa because the degree of load imbalance increases as the compute power gap between GPUs increases. For instance, Metis improves the performance 1.66×, 1.69×, and 2.15× compared to Alpa in P100+T4, T4+V100, and P100+V100 combinations, respectively, where the performance gap between P100 and V100 is the highest.

Different model architectures. Depending on model architectures, we have found that various factors contribute to the performance gains against other baselines, which shows the validity of our heterogeneity-aware search space.

For GPT-3, Metis outperforms other baselines 1.89× on average across all GPU combinations. The major factor for performance improvement is Metis's hetero-aware load balancing. While both AMP and Alpa partition layers across heterogeneous GPUs as if the GPUs are homogeneous, Metis assigns more layers to the faster GPUs by being aware of GPU heterogeneity. In detail, in T4 + V100 16 GPU experiment of GPT-3 (Fig. 4 (a)), Alpa selects a plan that consists of four stages, each of which contains 6 layers, but Metis partitions layers into four stages where 36% of the total layers are assigned to two stages mapped to slower T4 device groups.

In MoE, not only the layer load balancing but also other factors contribute to the performance gap between Metis and the existing work. For example, in T4 + V100 16 GPU experiments, although Metis and Alpa have the same layer partitioning ratio between T4 and V100 (both allocate 75% of layers to V100), Metis outperforms Alpa by 1.32×. In this case, Alpa allocates layers into three stages where the first stage consists of a device group with 4xV100 and 4x T4 GPUs, and its intra-stage plan is DP-only (DP=8). Within the device group, T4 GPUs become straggler in DP all-reduce due to their uniform batch split, leading to the idle time of V100 GPUs. On the other hand, by being aware of heterogeneous GPUs, Metis avoids stragglers within a device group by partitioning layers into five stages, each of which is executed on a device group consisting of a single type of GPU within the same node. In this way, without the need for additional load balancing within a stage, the plan of Metis outperforms that of Alpa.

For Wide-ResNet, Metis shows the largest performance gaps against the baselines compared to GPT-3 and MoE. The average performance gap is 2.93× on Wide-ResNet, while it is 1.89× and 2.42× on GPT-3 and MoE, respectively. Unlike GPT-3 and MoE, both of which are homogeneous models with repeated Transformer layers, WideResNet is a heterogeneous model where each layer has distinct compute demands.

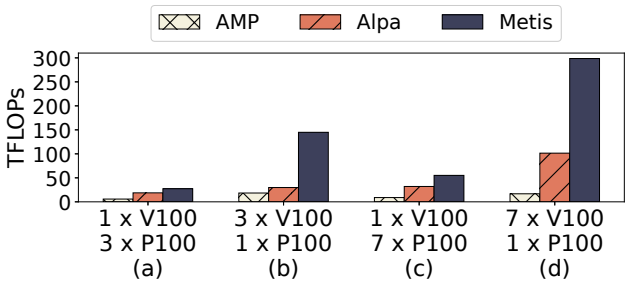

Figure 5: Performance on non-uniform GPU compositions where the number of GPUs for each GPU type varies.

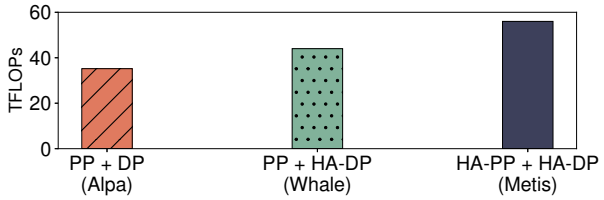

Figure 6: A microbenchmark to evaluate the effect of load balancing. Metis supports hetero-aware load balancing for both data and layers, represented as HA-DP and HA-PP in this graph, respectively (rightmost). Whale [17] only supports hetero-aware data load balancing (middle).

Specifically, our profiling results show that the rear layers of the model have higher compute demands: among 50 layers, the layer execution time (forward + backward) increases steeply after the 40^{th} layer. Such heterogeneity in the model architecture complicates the planning in the presence of GPU heterogeneity, and therefore, existing work does not find efficient parallelism plans.

In the WideResNet experiments with 8 T4+V100 GPUs, Metis shows 3.66× and 2.66× speedup against AMP and Alpa, respectively. The best plan of AMP only uses data parallelism, as the 2B Wide-ResNet model fits in a single GPU for both T4 (14 GB) and V100 (15 GB). In contrast, Metis partitions the layers into two stages, assigning compute-intensive rear layers to faster V100 GPUs. Each stage uses full DP (DP=4) in this plan. The speedup of 3.66× against AMP indicates that even for models that fit into a single GPU, DP-only plans can be suboptimal and leveraging pipeline parallelism with proper load balancing can be more favorable. Additionally, we have observed that Alpa leverages the same parallelism plan it finds as optimal in a homogeneous GPU environment, which is suboptimal in heterogeneous settings.

5.2.2 Non-uniform GPU composition

We also evaluate Metis on environments where each node has a different (non-uniform) number of GPUs. We use the GPT-3 model with P100 and V100 GPUs for this evaluation. Fig. 5 shows the experimental results on scenarios where either weak (P100 in (a) and (c)) or strong (V100 in (b) and (d)) GPUs are dominant in the cluster. Considering the compute power gap between V100 (≃ 7.9 x P100) and P100, plans on environments where stronger GPUs (V100) are dominant show superior performance for all baselines ((a) < (b) and (c) < (d) in Fig. 5). Throughout the experiments, AMP and Alpa show far less throughput than Metis (on average 9.1× and 2.7× speedup, respectively).

One of the main factors that improves the performance on non-uniform GPU composition is that Metis can compose diverse device groups in such environments, whereas others have a $M \times N$ node constraint. As a result, Metis can compose the non-uniform number of GPUs with a smaller number of nodes than AMP and Alpa, which can mitigate the network bottleneck across nodes during distributed training. For this evaluation, we use nodes with one GPU, two GPUs, and four GPUs. Consequently, Metis is able to compose Fig. 5 (a) and (b) GPU compositions only with three nodes—1x1 A, 1x1 B, and 1x2 B*—whereas AMP and Alpa are inevitable to compose the Fig. 5 (a) and (b) GPU compositions with four nodes—1x1 A and $3 \times 1 \times$B, to satisfy the $M \times N$ node constraint (M is 4 and N is 1). Similarly, Metis uses 4 nodes to compose Fig. 5 (c) and (d) GPU compositions (using 1x1 A, 1x1 B, 1x2 B, and 1x4 B GPUs), whereas AMP and Alpa require 8 nodes—1x1 A and 7x1 B GPUs (M is 8 and N is 1).

In this evaluation, regardless of the environment, AMP employs only one (Fig. 5 (a)) or two stages (Fig. 5 (b) – (d)), where all two-stage plans have uniformly split layers and all-DP intra-stage plans. Considering either four or eight nodes are used in AMP experiments, such device group and stage configuration could incur huge overheads for inter-node DP communication through inter-node network bandwidth. Differently from AMP, Alpa splits stages in the node boundary for most cases (e.g., 4 stages in (b) and 7 stages in (d), where 4 and 8 nodes are used, respectively), minimizing inter-node communication of DP and TP communications. However, Alpa still partitions layers without consideration of different heterogeneous GPUs as the planner harnesses the average of the profiled metrics, limiting its performance. On the other hand, Metis reduces the number of stages with three nodes for (a) and (b), and with four nodes for (c) and (d) to further minimize PP communication overheads while harnessing intra-stage parallelism across high intra-node network bandwidth. Moreover, Metis flexibly balances loads considering the compute capacity regardless of whether strong or weak GPUs are dominant.

5.3 Effectiveness of Load Balancing

We show the effectiveness of our search algorithm in terms of load balancing of both layers and data with a microbenchmark. Our microbenchmark consists of two baselines: (1)

*A and B is either V100 or P100

hetero-unaware load balancing like Alpa [49] that balances the load across GPUs assuming homogeneous GPUs (leftmost in Fig. 6) and (2) hetero-aware data partitioning like Whale [17] that leverages memory-constraint load balancing algorithm (middle in Fig. 6).

To show the effectiveness of both layer and data load balancing, we create two stages, each of which has heterogeneous GPUs. The first stage contains 1x1 V100 + 1x1 T4 GPUs, and the second stage includes 1x1 T4 + 1x1 P100 GPUs. We use the GPT-3 350M model with a global batch size of 16 for the microbenchmark, and the model is partitioned into two stages. In this case, Metis can balance the micro-batch size within the first stage and layers across the stages based on the computing capacity of heterogeneous GPUs, the execution cost of layers, and the peak memory usage.

Fig. 6 shows the effectiveness of both layer and data load balancing. Compared with Whale's and Alpa's approach, Metis achieves 19% and 22% performance gains, respectively. In the heterogeneous environment where both intra-stage and inter-stage device composition can be heterogeneous, load balancing of both data and layers is required to prevent stragglers from slowing down the training. To this end, the load balancing of Whale [17] for heterogeneous devices is limited. Their load balancing considering heterogeneous compute and memory is only confined to load balancing within a single stage (intra-TaskGraph load balance in their term), which is a subset of our algorithm. In the meantime, their inter-stage load balancing (inter-TaskGraph in their term) does not balance layer allocation but focuses on device placement considering uneven memory demands of each stage. In contrast, our layer allocation mechanism generally covers such cases while guaranteeing balanced load distribution of layers across heterogeneous GPUs.

5.4 Profiling and Searching Overhead

Due to the huge search space, it is crucial to reduce the overhead of profiling metrics and searching plans. Table 4 shows the normalized profiling and searching overheads of Oracle compared to Metis, where Oracle profiles all possible combinations of layer execution time and memory usage for each stage and GPU type, while exhaustively searching all possible plans. Compared to Oracle, Metis significantly reduces these costs by orders of magnitude. In the evaluation on 4~64 GPUs, Metis spends 12.5, 13.3, 15.5, 18.2, and 18.9 minutes for profiling and 0.1, 0.5, 1.7, 58.9, and 600 seconds for searching in the experimental environment. Since Metis profiles repetitive layers only once for each GPU type, the profiling time does not significantly increase as the number of GPUs and the model size increases. The search time in Metis compared to Oracle is also significantly reduced as the number of in-use GPUs increases: from 17× to 5M×. This effectiveness is primarily because Metis filters out a large number of inefficient plans. In addition, the profiling and searching costs of Metis

	Profiling	**Search**	
#GPUs	**Oracle**	**Oracle**	**Metis$^{\alpha}$**
4	96	17	17
8	228	108	108
16	270	480	327
32	> 3171	> 29K	790
64	> 3294	> 5M	1100

Table 4: The *normalized* profiling and search overheads of Oracle and the variant of Metis for GPT-3 with T4+V100 GPUs (normalized by Metis). To break down the effectiveness of Metis's searching techniques, we compare the search cost of Metis without capacity-aware load balancing and DP-first DFS search (**Metis$^{\alpha}$**) by exploring full layer-partitioning plans and intra-stage parallelism. Metis prunes stage-device group pairs with 0.5 *var* and the maximum permutation length 10.

are significantly lower than those of Alpa, while finding better plans on heterogeneous GPUs. For example, on 32 and 64 GPUs, Alpa spends 105 and 209 minutes on profiling and 141 and 240 minutes on searching, respectively, which requires 5.8× and 11× more profiling time and 143× and 24× more searching time than Metis.

To validate the efficacy of each core technique in Metis, we also compare Metis with a search algorithm that applies only the pruning of stage-device group pairs, which is represented as Metis$^{\alpha}$. The effectiveness of stage-device pair pruning is estimated by comparing Oracle and Metis$^{\alpha}$. In 4 and 8 GPUs, Metis$^{\alpha}$ explores all possible stage-device pairs like Oracle. However, in ≥16 GPUs, Metis$^{\alpha}$ filters out inefficient plans such as high variance number of GPUs per stage (e.g., stage 1 has one GPU, but stage 2 has 8 GPUs) and similar combinations (e.g., in 16 GPUs with 15 number of stages, permutations of 14 stages with one GPU and one stage with two GPUs), which are not selected as the best plan. Similarly, comparing Metis$^{\alpha}$ with Metis shows the effectiveness of capacity-aware load balancing and efficient intra-stage planning with DP-first DFS searching.

Even with this massive reduction in profiling and searching overheads, Metis is still able to find near-optimal plans. We compare the performance of plans generated by Metis and Oracle in Table 4. For the experiments with 4 and 8 GPUs, Metis outputs the same parallel plan as Oracle. For the experiment with 16 GPUs, layer partitioning of the two plans slightly differs, leading to a marginal performance gap. Specifically, both systems use four stages where only TP is used for the intra-stage plans. Due to memory constraints, increasing the DP degree within each stage causes an OOM exception in this experiment. Regarding the layer partitioning, the first stage of Metis contains one more layer than that of Oracle while having one less layer in the last stage. In 32 and 64 GPUs, we could not automatically find the optimal plan within tractable

time because it takes a huge amount of time for profiling and searching (more than 200 hours). Instead, to check whether there exist better plans than the one selected by Metis, we manually adjust the Metis's plan multiple times by slightly changing the layer load balancing, the number of stages, stage-device group pairs, and so on. However, most of them lead to slower performance than Metis, while only few of them lead to slightly higher performance than Metis (around 5%). These results indicate that even if Metis filters out the best plan, Metis finds near-optimal plans similar to the best one. Importantly, such difference is immediately overshadowed by the huge profiling and search overhead of Oracle, and we will further discuss the difference in § 7.

6 Related Work

DNN training on heterogeneous infrastructure. There are several works to execute a distributed DL training job on heterogeneous infrastructure [6, 9, 10, 17, 25, 32, 46]. Some of the work has a limited search space, such as exploring only DP and TP [46,48], only TP [39], or batch-size adjustment [17,50] on heterogeneous GPUs, without the exploration of PP and layer partitioning across stages that is the important factor for distributed training of large-scale models. SDPipe [25] develops a framework for dynamic heterogeneity where peak GPU FLOPS are dynamically changed during runtime. HPH [10] and HetPipe [32] optimize DP and PP on heterogeneous GPUs (without TP), but they do not explore how to partition layers across stages and other parameters. Metis is the first system that fully searches all parameters described in § 4.1 and automates parallelism plans with an efficient search algorithm and cost model based on lightweight profiling.

Auto-tuning parallelism on homogeneous GPUs. We already compare AMP [21] and Alpa [49] in our evaluation. There are several works to automate tensor or operator-level parallelisms on homogeneous GPUs [18, 40, 43]. Merak [19] automates 3D parallelism for giant foundation models, and Galvatron [26] also automates DP, TP, and PP in homogeneous GPUs for Transformer-based models. In contrast, Metis's automatic decision is agnostic to models and is optimized on heterogeneous GPUs with a huge search space.

Cluster schedulers on heterogeneous GPUs. To reduce the waiting time of training jobs in heterogeneous GPU clusters, existing work proposes optimizing cluster-level schedulers [5, 11, 15, 24, 27, 30, 33, 45]. Sia [15] and Hare [7] optimize the scheduling of multi-jobs and as well as optimizing intra-job parallelisms (e.g., increasing data parallelism) on heterogeneous GPU clusters. All of these works execute a single deep learning job on homogeneous GPUs while packing multiple training jobs as much as possible within the heterogeneous GPU clusters, which is orthogonal to Metis.

7 Discussion

Dynamic plan adaptation. Metis finds near-optimal parallelism plans within a short time using an efficient search algorithm, but there is a possibility of missing the best plan compared to the Oracle approach. Although we have observed that the plan selected by Metis has negligible differences from the best plan selected by the Oracle approach in our evaluation, and the actual training time of one epoch shows negligible differences, such differences may lead to a large gap in the total training time when the number of epochs is significantly large. In the future, we can address this problem with the dynamic adaptation of plans. To quickly execute distributed training, Metis can run the efficient search algorithm and find a near-optimal decision. During the training, Metis can run the exhaustive search algorithm to find the best plan and adjust the near-optimal plan to the best one if they are different. The dynamic adaptation is orthogonal to Metis, which is an interesting topic for future work.

Various GPU types and multi-vendor GPUs. Although this work evaluates Metis on single-vendor (NVIDIA) heterogeneous GPUs with three different types (P100, V100, and T4), we believe that Metis generally finds better parallelism plans than others in various GPU environments because heterogeneity-aware device grouping and load balancing based on profiling are generally applicable regardless of tje types of GPUs. In multi-vendor GPU environments, a major limitation of training a model is compatibility. Supporting compatibility between different vendor GPUs is another interesting and promising research topic, which is orthogonal to Metis.

8 Conclusion

We design and implement Metis, a complete system for automating distributed deep learning training by exploring a vast hetero-aware search space, including the number of stages, diverse device groups, intra-stage parallelism, load balancing strategies, and the number of micro-batches. To efficiently explore the expanded search space, Metis develops a new efficient search algorithm, which explores diverse resource trade-offs between parallelism plans with capacity-aware load balancing and DP-first strategies. Metis further reduces the profiling overheads by exploiting the characteristics of models with a cost model for estimating the execution time and the memory usage of layers. Our evaluation shows that Metis finds better parallelism plans than state-of-the-art with lower profiling and searching overheads, improving the training speed by 1.05 to 8.43×.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. We also gratefully acknowledge Yunsu Lee for helping us to initiate and complete this work.

References

[1] Inside Volta: The World's Most Advanced Data Center GPU. https://developer.nvidia.com/blog/inside-volta/.

[2] NVIDIA Blackwell Architecture. https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/.

[3] NVIDIA Hopper Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 1877–1901, 2020.

[5] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys 20)*, pages 1–16, 2020.

[6] Chen Chen, Qizhen Weng, Wei Wang, Baochun Li, and Bo Li. Semi-dynamic load balancing: Efficient distributed learning in non-dedicated environments. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC 20)*, pages 431–446, 2020.

[7] Fahao Chen, Peng Li, Celimuge Wu, and Song Guo. Hare: Exploiting inter-job and intra-job parallelism of distributed machine learning on heterogeneous gpus. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC 22)*, pages 253–264, 2022.

[8] Jeffrey Dean, Greg S Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V Le, Mark Z Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, et al. Large scale distributed deep networks. In *International Conference on Neural Information Processing Systems (NeurIPS 12)*, pages 1223–1231, 2012.

[9] Yifan Ding, Nicholas Botzer, and Tim Weninger. Hetseq: Distributed gpu training on heterogeneous infrastructure. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 15432–15438, 2021.

[10] Yabo Duan, Zhiquan Lai, Shengwei Li, Weijie Liu, Keshi Ge, Peng Liang, and Dongsheng Li. Hph: Hybrid parallelism on heterogeneous clusters for accelerating large-scale dnns training. In *2022 IEEE International Conference on Cluster Computing (CLUSTER 22)*, pages 313–323. IEEE, 2022.

[11] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[12] Marius Hobbhahn and Tamay Besiroglu. Trends in GPU price-performance. https://epochai.org/blog/trends-in-gpu-price-performance.

[13] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems (NeurIPS 19)*, 32, 2019.

[14] Buffer donation in jax. https://jax.readthedocs.io/en/latest/faq.html#buffer-donation, 2023.

[15] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 23)*, page 642–657, 2023.

[16] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant gpu clusters for training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[17] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. Whale: Efficient giant model training over heterogeneous GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 673–688, 2022.

[18] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems (MLSys 19)*, 1:1–13, 2019.

[19] Zhiquan Lai, Shengwei Li, Xudong Tang, Keshi Ge, Weijie Liu, Yabo Duan, Linbo Qiao, and Dongsheng Li. Merak: An efficient distributed dnn training framework with automated 3d parallelism for giant foundation

models. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1466–1478, 2023.

[20] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations (ICLR 20)*, 2020.

[21] Dacheng Li, Hongyi Wang, Eric Xing, and Hao Zhang. Amp: Automatically finding model parallel strategies with heterogeneity awareness. *Advances in Neural Information Processing System (NeurIPS 22)*, 35:6630–6639, 2022.

[22] Mingzhen Li, Wencong Xiao, Hailong Yang, Biao Sun, Hanyu Zhao, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, et al. Easyscale: Elastic training with consistent accuracy and improved utilization on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 23)*, pages 1–14, 2023.

[23] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2391–2404, 2023.

[24] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient {GPU} cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[25] Xupeng Miao, Yining Shi, Zhi Yang, Bin Cui, and Zhihao Jia. Sdpipe: A semi-decentralized framework for heterogeneity-aware pipeline-parallel training. *Proceedings of the VLDB Endowment*, 16(9):2354–2363, 2023.

[26] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *VLDB Endowment*, 16(3):470–479, nov 2022.

[27] Zizhao Mo, Huanle Xu, and Chengzhong Xu. Heet: Accelerating elastic training in heterogeneous deep learning clusters. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 499–513, 2024.

[28] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 19)*, pages 1–15, 2019.

[29] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning (ICML 21)*, pages 7937–7947. PMLR, 2021.

[30] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.

[31] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 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 (SC 21)*, pages 1–15, 2021.

[32] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young ri Choi. HetPipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 307–321, 2020.

[33] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18, 2021.

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

[35] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 21)*, pages 1–14, 2021.

[36] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 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 (KDD 20)*, pages 3505–3506, 2020.

[37] Noam Shazeer, *Azalia Mirhoseini, *Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations (ICLR)*, 2017.

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

[39] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Accpar: Tensor partitioning for heterogeneous deep learning accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA 20)*, pages 342–355. IEEE, 2020.

[40] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, 2022.

[41] Profiling in alpa. https://github.com/alpa-projects/alpa/blob/main/alpa/pipeline_parallel/stage_profiling.py#L587.

[42] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Connor Holmes, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. Zero++: Extremely efficient collective communication for giant model training. *arXiv preprint arXiv:2306.10209*, 2023.

[43] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019 (Eurosys 19)*, pages 1–17, 2019.

[44] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. {MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, 2022.

[45] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[46] 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 *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 20)*, pages 93–107, 2020.

[47] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[48] Shiwei Zhang, Lansong Diao, Chuan Wu, Zongyan Cao, Siyu Wang, and Wei Lin. Hap: Spmd dnn training on heterogeneous gpu clusters with automated program synthesis. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys 24)*, pages 524–541, 2024.

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

[50] Xin Zhou, Ling Chen, and Houming Wu. Abs-sgd: A delayed synchronous stochastic gradient descent algorithm with adaptive batch size for heterogeneous gpu clusters. *arXiv preprint arXiv:2308.15164*, 2023.

[51] Martin A Zinkevich, Markus Weimer, Alex Smola, and Lihong Li. Parallelized stochastic gradient descent. In *International Conference on Neural Information Processing Systems (NeurIPS 10)*, pages 2595–2603, 2010.