Ditto: Automating Hybrid Parallelism for Large Model Training on Heterogeneous Cluster

Zhenyan Zhu [*] **Lai Wang** **Parin Senta** **Hithesh Prabhakar Reddy** **Noah Kaplan**

Abstract

Ditto introduces an efficient method for searching near-optimal hybrid-parallel combinations for training large models on heterogeneous GPU clusters. Traditional distributed training systems exhaustively search through different parallelism combinations within homogeneous clusters, which is feasible due to a limited search space. However, when applied to heterogeneous clusters, where GPU types vary, the search space expands exponentially with the number of node types, making exhaustive search impractical. Ditto overcomes this by approximating the optimal pipeline template(a configuration that partitions a model using hybrid parallelism) within a polynomial runtime. Ditto achieves this in three main steps: First, it "folds" the heterogeneous cluster into a logically homogeneous one based on the fact that DNN execution time on a hardware configuration is predictable (Qi et al., 2022)(Gujarati et al., 2020). Second, it partitions the model's layers into uniform stages, with each stage assigned to one logical device. Finally, Ditto uses a dynamic programming approach to convert the virtual template back into a real pipeline template based on resource constraint. The optimal combination of hybrid parallelism is explored in the final step. Our evaluations demonstrate that Ditto can quickly identify nearly optimal templates for large models across various heterogeneous cluster configurations, significantly outperforming exhaustive search algorithms.

1 Introduction

Deep Neural Networks (DNNs) are increasingly getting larger(Villalobos et al., 2022). As the by-product of those large models, the training of these models can be extremely expensive. As a result, there are a lot recent advances in training systems (Shoeybi et al., 2019), (Cai et al., 2021), (Huang et al., 2019), (Zheng et al., 2022), (Jang et al., 2023) scaling large model training by leveraging hybrid parallelism, namely the combination of tensor-level parallelism, pipeline-level parallelism, and data-level parallelism. These systems deliver an order of magnitude in training performance by exhaustively searching the best combination of these parallelism. However, they made the assumption that the underlying cluster is homogeneous, and the usage of heterogeneous GPUs in data centers is inevitable due to the short release cycle of new GPU architectures (Jiang et al., 2017). Based on the GPU performance trend report from Epoch AI(Hobbhahn & Besiroglu, 2022) as shown in figure 1, the performance of GPUs would be doubled every 2.31 year approximately. When data centers buy in those powerful devices, it is necessary to collaborate them with the old devices for achieving maximum resource utilization.

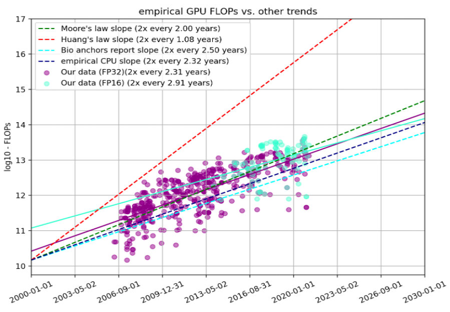


Figure 1. Epoch AI's analysis of GPU FLOP/s vs release date for their dataset and relevant trends from the existing literature

Unfortunately, supporting heterogeneity in cluster adds up another level of complexity in those systems to find the optimal hybrid parallelism. This is because the core algorithm in those systems need to iterate on all possible combinations of device-workload mapping and gradually construct an optimal combination of mapping the whole model on the cluster(Zheng et al., 2022)(Jang et al., 2023). For a homogeneous cluster, as the device type is the same, we can simply use a tuple $T = (G, d)$ to denote the optimal pipeline template, meaning the best parallel configuration of mapping a subset of the model's computation graph G on d devices. However, for a heterogeneous cluster with m types of devices, we might need to use a more complicated representation, $T = (G, [d1, ..., d_m])$ to represent the pipeline template. It's trivial that iterating on all possible subsets of the combinations will lead to a search space that is exponential to m.

To reduce the complexity of the search space, we propose a novel approach called "node folding" to convert the heterogeneous cluster with $(d_1, ..., d_m)$ devices into a logical homogeneous cluster with d devices. This can be done based on the fact that DNN training or inference is fundamentally a deterministic sequence of mathematical operations that has a predictable execution time given a hardware configuration(Qi et al., 2022)(Gujarati et al., 2020).

We have implemented Ditto on top of Oobleck, which implements a pipeline template generation algorithm using divide and conquer, supporting PyTorch models and HuggingFace Transformers.

Overall, we made the following contributions:

- An extension of Oobleck's template generation algorithm to support exhaustively finding the optimal pipeline template on heterogeneous clusters and an analysis of the complexity of the extended algorithm.
- We present Ditto, an extensible library that gives Oobleck the illusion that the underlying cluster is homogeneous. Ditto will then reconstruct the real template from the virtual template generated by Oobleck.
- We present a cost-modeling technique for collecting the computation and communication cost of any arbitrary PyTorch layers by utilizing XLA's cost model(Sabne, 2020).
- We evaluate Ditto's efficiency and optimality with GPT2XL on several configurations of heterogeneous clusters by comparing with the optimal planning algorithm extended from Oobleck.

2 Background

In this section, we briefly introduce three levels of parallelism commonly used for large model training: data-level parallelism, pipeline-level parallelism, and tensor-level parallelism. Hybrid parallelism is the combination of those.

2.1 Data-level parallelism

In data-level parallelism, the batched training data is distributed across different workers, and each worker owns a replica of the whole model. This requires the GPUs assigned to the worker to hold the entire model(Hwang et al., 2021)

2.2 Tensor-level parallelism

Tensor-level parallelism refers to the approaches to partitioning individual operators/layers(eg. matmul) across different devices. It assigns each chunk of the tensor to a designated GPU. Each GPU independently works on its assigned chunk, allowing for parallel computation across multiple GPUs. However, communications(all-reduce, all-to-all) are required when data dependency exists across multiple GPUs.

2.3 Pipeline-level parallelism

Instead of partitioning individual layers which may require a significant amount of communications, pipeline-level parallelism will group several layers into a stage that will be assigned into a stage. it also splits the training batch as many microbatches and pipelines the forward and backward passes across microbatches on distributed workers. Properly partitioning the model into several pipelines can significantly increase the throughput of training/inference. However, too many stages will lead to a long pipeline drain time.

3 Extending Pipeline Generation Algorithm in Oobleck for Heterogenous Cluster

In section 4.1.2 of Oobleck(Jang et al., 2023), Jang et.al presents a pipeline template generation algorithm that employs a divide-and-conquer strategy: it will divide the model into pipeline stages and iterates over all possible combinations of GPU–stage mapping to find the one that minimizes the iteration time across a distributed homogeneous GPU cluster setup. Within each stage, data-level parallelism and tensor-level parallelism would be used. We extend the algorithm so that it can exhaustively search over the combinations of different types of GPUs and stages for finding the global optimal partition template on a heterogeneous GPU cluster. Let $T(S', u, v, x)$ be the minimum pipeline iteration time for layers $(l_u, l_{u+1}, ...l_{v-1})$ partitioned into S' stages running on x(cluster spec), where $x = (d_1, ..., d_m)$, meaning that there is d_i number of device for device type $i, 1 <= i <= m$. The pipeline for the entire model using all heterogeneous GPUs has the minimum iteration time

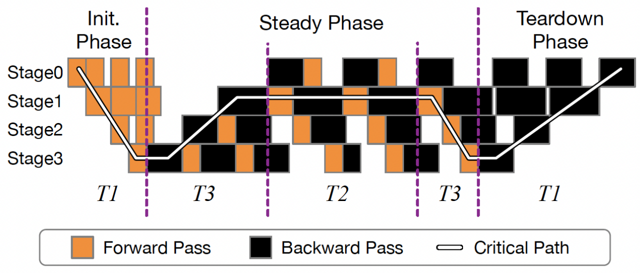


Figure 2. Model training pipeline iteration time breakdown. Cited from Oobleck (Jang et al., 2023)

$T(S, 0, L, (M * n_1, ..., M * n_m))$, where the model has L Layers, and there are M GPUs within each node and n_i nodes for device type i.

We adopted Oobleck's methodology in calculating the minimum iteration time of a training pipeline as shown in figure 2. $T1$ denotes the pipeline initialization and draining time, and $T2, T3$ represent the time taken in the pipeline's steady state. Please refer to section 4.1.2 of Oobleck (Jang et al., 2023) for a more detailed problem definition. $T1$, $T2$, and $T3$ have the following definition in the division phase after extending to the heterogeneous cluster setting: (k^* is the index to the slowest stage in the pipeline)

$$\begin{aligned} &T_{1,s,k,x'}(S', u, v, x) \\ &= T_{1,s,k,m}(s, u, k, x^{'}) + T_{1,s,k,m}(S' - s, k+1, v, x - x^{'}) \end{aligned} \tag{1}$$

$$\begin{aligned} &T_{2,s,k,x'}(S', u, v, x \mid k^*) \\ &= (N_b - S' + k^* - 1)(F_{s_{k^*,m}} + B_{s_{k^*,m}}) \end{aligned} \tag{2}$$

$$\begin{aligned} &T_{3,s,k,x'}(S', u, v, x \mid k^*) = \\ &\begin{cases} T_{3,s,k,x'}(s, u, k, x^{'} \mid k_1) & \\ +T_{1,s,k,x'}(S' - s, k+1, v, x - x^{'}) & \text{if } k^* = k_1 \\ T_{3,s,k,x'}(S' - s, k+1, v, x - x^{'} \mid k_2) & \text{else} \end{cases} \end{aligned} \tag{3}$$

Let x and x' be defined as:

$$x = \{(d_1, \ldots, d_m)\}$$

$$x' = \{(d'_1, \ldots, d'_m)\}$$

such that $0 \leq d'_i \leq d_i$ for all $1 \leq i \leq m$. The element-wise subtraction of x' from x is then given by:

$$x - x' = \{(d_1 - d'_1, \ldots, d_m - d'_m)\}$$

We iterate over s, k and all possible configurations $x^{'}$ of x to find a $(s, k, x^{'})$ that minimizes $T_{1,s,k,x'} + T_{2,s,k,x'} + T_{3,s,k,x'}$. The conquer phase and other details are the same as the problem definition in section 4.1.2 of Oobleck(Jang et al., 2023). Please refer to that section for more details.

Complexity Analysis: Given that the complexity of Oobleck's planning algorithm for finding the optimal template on a homogeneous cluster setting is $O(L - n)L^3 nM$, where the nM part is composed by $O(n)$ for partitioning n homogeneous nodes and $O(M)$ for partitioning M GPUs within a single node. Our algorithm extends this part by enumerating all possible configurations $x^{'}$ of x. To analyze the complexity of the enumeration, we consider that for each device count d_i in x, there are $d_i + 1$ possible values, so the complexity for enumerating the tuple $(d_1, ...d_m)$ is $(d_1 + 1) * ... * (d_m + 1)$ given that each element is independent of each other. Since $d_i <= M * n_i$, the complexity for enumerating all possible $x^{'}$ is $O((n_{max}M + 1)^m)$, where $n_{max} = max_{i=1}^{m}(n_i)$. The overall complexity of finding the optimal template for a heterogeneous cluster setting is

$$\begin{aligned} &O((L - n)L^3 * O(enumerate(x^{'})) \\ &= O((L - n)L^3(n_{max}M + 1)^m) \end{aligned} \tag{4}$$

given $n = sum_{i=1}^{m}(n_i)$

4 DITTO OVERVIEW

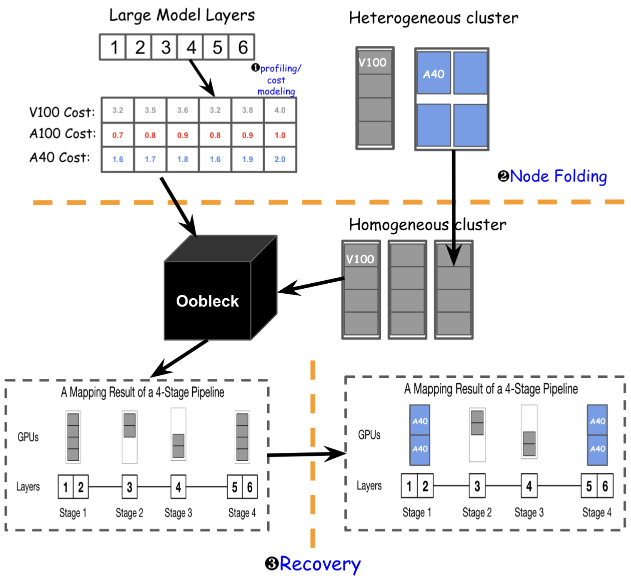


Figure 3. Ditto Overview

To prevent the search space from scaling exponentially, we present Ditto to give Oobleck an illusion of searching over a homogeneous cluster and reconstruct the optimal template using a dynamic programming algorithm with polynomial runtime. Ditto is an extensible and modular library that contains three main components: (1) a cost modeling tool that can gather the PyTorch model's layer's computation/communication cost from XLA's cost model(Sabne, 2020); (2) a node folding library that can convert a heterogeneous GPU cluster to a "representative" homogeneous GPU cluster based on the model's overall cost collected from (1). We then leverage Oobleck's template generation algorithm to generate a pipeline template plan based on the virtual homogeneous cluster. Finally, (3) a recovery solver containing a greedy algorithm and a bottom-up dynamic programming algorithm will reconstruct a ”near-optimal” template based on the device constraints

5 Node Folding

Previous research (Qi et al., 2022)(Gujarati et al., 2020) illustrates that DNN training or inference is fundamentally a deterministic sequence of mathematical operations that has a predictable execution time given a hardware configuration. This fact allows us to establish a comparative performance model between different GPU devices. For instance, if in a heterogeneous cluster containing one GPU A and one GPU B, a DNN model executes three times faster on B than on A, we can predict the model's cost on this cluster is equivalently as four GPU A, assuming the communication cost within one node is negligible. This method enables the approximation of a homogeneous cluster using a heterogeneous one. In section 5.3, we will also propose a node folding algorithm considering the communication cost.

To construct the “node folding” model, it is necessary to obtain the execution cost of the whole model for each device type. We achieve this by profiling/cost modeling the execution cost of each layer in the model across all device types in the cluster and aggregating all layers to get the whole model's cost because (1) Oobleck also requires each layer's cost for finding the optimal plan and (2) the whole model may not be able to fit into the device's memory capacity for profiling. Additionally, we also need a mathematical model to normalize device resources based on the “weakest” device type, the type which has the most cost for the given workload. The normalization is based on the “weakest” device type because it can give us maximum flexibility in plan recovery algorithm proposed in section 6.

The profiling/cost modeling methodology and node folding mathematical model will be discussed in the below subsection. An example of the node folding pipeline is shown in figure 5. The generated homogeneous cluster and the layer cost on the ”weakest” device type will become the input for existing Oobleck's planning system(Jang et al., 2023). It will then generate a list of optimal pipeline templates with a different number of stages. Section VI will discuss the methodology of how we can reconstruct the relative optimal pipeline template based on the heterogeneous cluster device constraints.

5.1 Profiling

With access to the heterogeneous cluster, profiling could be done to estimate layer execution cost for node folding. For each type of node, one of them is picked to be profiled. Profiling is done by running the forward and backward passes for each layer on the first device of that node. Forward time, backward time, and memory needed for each layer are encoded as $device_i :: layer_j.forward$, $layer_j.backward$, $layer_j.memory$ for i in $1...m$ and j in $1...L$, where m is the total number of device types and L is the total number of layers. Additionally, the communication cost within node on d devices for each layer is collected by profiling the time of an all-reduce operation on d devices, encoded as $device_i :: layer_i.c\_in[d]$, for d in $1...M$ and M is the total number of device within each node. For simplicity, communication costs across nodes $device_i :: layer_i.c\_accross$ are not considered in our profiling routine because Oobleck does not consider the communication cost across nodes in the pipeline generation algorithm.

5.2 Cost Modeling

To gather the execution cost of each layer without access to a real heterogeneous cluster, we leverage the analytical model found in the OpenXLA(Sabne, 2020). This requires compilation from PyTorch layer objects to HLO modules, the intermediate representation native to the XLA ecosystem. we developed an end-to-end lowering pipeline to query the execution cost given any arbitrary PyTorch layer and any arbitrary hardware configuration following OpenXLA's style as shown in figure4: it starts with lowering the PyTorch graph to StableHLO, an ML opset dialect in MLIR(Lattner et al., 2021), through torch-mlir(tor, 2023) and then converts it to HLO module and passes to the XLA cost model. XLA cost model will visit each operator in the ML graph and accurately record the flops and bytes used. It will calculate the computing time, memory read/write time, and communication time based on the provided hardware configuration and generate the overall execution time for the ML graph. We construct the $device :: layer$ structure required in Oobleck and fill the fields proposed in section 5.1 correspondingly

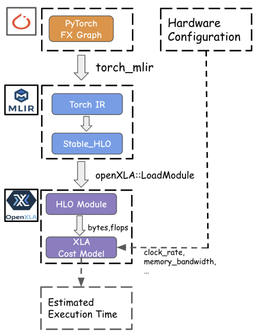


Figure 4. Use XLA cost model for PyTorch Models

5.3 Node Folding Algorithm

The main goal of the node folding algorithm is to find a homogeneous cluster with n nodes such that it has approximate cost as the heterogeneous cluster with $n_1, n_2, ..., n_m$ nodes with m types of devices for the specific input workload. Each node in heterogeneous cluster or homogeneous cluster will contain M device.

After profiling or querying the cost model of each layer's cost, we get each layer's execution cost on each type of device $c_i(l_j) = device_i :: layer_j.forward + device_i ::$

$layer_j.backward$ for i in $1...m$ and j in $1...L$, where m is the total number of device types and L is the total number of layers. The "weakest" device type w is the type that the layer's execution cost is maximized across $1...m$. The model's aggregated cost on this "weakest" device is given by the equation:

$$total\_cost\_weak = max_i(\sum_{j=1}^{L}(c_i(l_j)))$$

To decompose this problem, we need to find the variable d_i , referring that for given workload, each device with type i has approximate cost of parallelizing d_i "weakest" device. Since each node in the cluster has M devices, we can use a tuple $(n_i^{'}, d_i^{'})$ to represent d_i in this cluster configuration, where $n_i^{'} = floor(\frac{d_i}{M})$ and $d_i^{'} = d_i - n_i^{'} * M$. This means that d_i devices is essentially $n_i^{'}$ node with M device in each node plus one node with $d_i^{'}$ device. Then we can aggregate d_i for M devices in n_i nodes across m types to find n:

$$n = \sum_{i=0}^{m} n_i * (M * n_i^{'} + round(\frac{(d_i^{'} * M)}{M})) \\ = \sum_{i=0}^{m} n_i * (M * n_i^{'} + d^{'}) \quad (5)$$

Section 5.3.1 and 5.3.2 will discuss how to find d_i ignoring communication cost and considering communication cost respectively.

5.3.1 Node folding ignoring communication cost

We define the term $p(d) = \frac{total\_cost\_weak}{d}$ to express the cost of the workload perfectly parallized using d "weakest" device with three levels of parallelism mentioned in section2. We ignore the overhead from pipeline-level parallelism because it would be removed when merging stages in the recovery algorithm(section6.2). To find the number of folded "weakest" device d_i for each device with type i, we need $p(d_i)$ to have approximate cost as the aggregated cost of the workload on device type i

$$\sum_{j=1}^{L}(c_i(l_j)) \approx p(d_i)$$

meaning

$$d_i = round(\frac{total\_cost\_weak}{\sum_{j=1}^{L}(c_i(l_j))})$$

We round the d_i to the nearest integer because we cannot generate fractional numbers of devices. In the dynamic programming algorithm presented in section6.2 and evaluation part, we will follow this heuristic in node folding. This is because the dynamic programming solution requires a pipeline template generated from Oobleck to have stage of length $n * M$, in which each stage will only be assigned to one device, so there is no within-node communication cost; across-node communication cost will also be ignored because the algorithm will merge d_i stages into one large stage.

5.3.2 Node folding considering communication cost

If we treat the recovery algorithm as a black-box, it is necessary to take communication cost into consideration while constructing node folding algorithm. Assuming the cost of across node communication is $c_{across}(d)$ and cost of within node communication is $c_{in}(d)$ on the weakest device type w, the definition of $p(d)$ would become

$$p(d) = \frac{total\_cost\_weak}{d} + c_{across}(d) + c_{in}(d)$$

We use the average within and across node communication cost among each layer collected from the profiler/cost model to approximate $c_{across}(d)$ and $c_{in}(d)$ because we don't know the layer assignment before running the planning system. The definition of average within and across node communication cost among the layers $l_{c\_in}[d]$ and l_{c_across} would be

$$l_{c\_in}[d] = \frac{\sum_{i=0}^{L}(device_w :: layer_i.c_{in}[d])}{L}$$

$$l_{c\_across} = \frac{\sum_{i=0}^{L}(device_w :: layer_i.c_{across})}{L}$$

Since d can be decomposed as $n^{'}$ nodes with M devices and one node with $d^{'}$ device

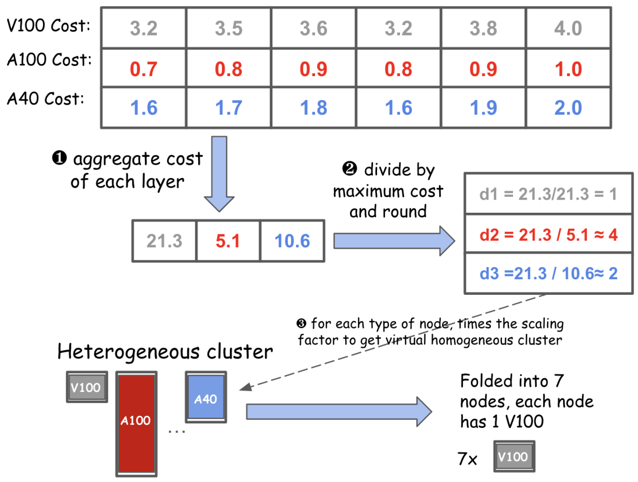
$$n^{'} = floor(\frac{d}{M}), d^{'} = d - n^{'} * M$$

We can approximate $c_{across}(d)$ as (total number of nodes $-1 = n^{'}$) $* l_{c\_across}$ and $c_{in}(d)$ as the number of layers allocated on $n^{'}$ nodes $*l_{c\_in}[M]+$ the number of layers allocated on one node with $d^{'}$ devices $*l_{c\_in}[d^{'}]$ because within node communication is required after executing each layer for simplicity. In reality, within node communication is only required when an input tensor of a layer does not satisfy the sharding spec of the chosen parallel algorithm for the layer(Zheng et al., 2022). We assume the number of layers allocated on $n^{'}$ nodes is $\frac{L*n^{'}*M}{n^{'}*M+d^{'}}$ and the number of layers allocated on one node with $d^{'}$ devices is $\frac{L*d^{'}}{n^{'}*M+d^{'}}$ for workload balancing. Formally, we define

$$c_{across}(d) = n^{'} * l_{c\_across}$$

$$c_{in}(d) = \frac{L * d^{'}}{n^{'} * M + d^{'}} * l_{c\_in}[d^{'}] + \frac{L * n^{'} * M}{n^{'} * M + d^{'}} * l_{c\_in}[M]$$

Based on the definition of $p(d)$, $c_{across}(d)$ and $c_{in}(d)$, we iterate d_i from $1...d_i^*$ to find the d_i such that the absolute error of $p(d_i) - \sum_{j=1}^{L}(c_i(l_j))$ is minimized. d_i^* is the number of folded devices without considering communication cost.



V100 Cost:
3.2
3.5
3.6
3.2
3.8
4.0
A100 Cost:
0.7
0.8
0.9
0.8
0.9
1.0
A40 Cost:
1.6
1.7
1.8
1.6
1.9
2.0
❶ aggregate cost of each layer
21.3
5.1
10.6
❷ divide by maximum cost and round
d1 = 21.3/21.3 = 1
d2 = 21.3 / 5.1 ≈ 4
d3 =21.3 / 10.6≈ 2
❸ for each type of node, times the scaling factor to get virtual homogeneous cluster
Heterogeneous cluster
V100
A100
A40
Folded into 7 nodes, each node has 1 V100
7x
V100

Figure 5. Node Folding end-to-end example ignoreing communication cost

6 Plan Recovery

Given a homogeneous cluster configuration composed of n "weakest" nodes with M devices in each node generated by the node folding algorithm proposed in section 5, we can input this configuration along with the model into Oobleck's pipeline generation routine. Oobleck will generates a list of pipeline templates with various length of stages from n to $n * M$. Each pipeline template contains optimal configuration of pipeline-level parallelism and tensor-level parallelism by minimizing the $T = T1 + T2 + T3$ explained in section3. Oobleck will return the one with the minimized cost among these templates. our focus is to find one of the virtual pipeline templates(may not be the optimal one) and recover the actual template for the heterogeneous cluster. In this section, we use the prefix "virtual" to refer to device resources and templates generated with the virtual homogeneous cluster and "real" for those applicable to the actual heterogeneous cluster.

6.1 Greedy Algorithm

A straightforward approach involves recovering the real template from the optimal pipeline template generated by Oobleck. Here, the model's partitioning is fixed and it maximizes the throughput of a cluster that includes only the "weakest" devices. We observed that the "strongest" device has the fewest number of stages to assign that can maximize the device utilization. For example, in the bottom part of figure 3, it's trivial that the only A100 should be assigned to stage1 or stage4; otherwise it will waste a lot of computing resource. To maximize device utilization, the most powerful devices should be allocated first to the stages in the virtual plan. Following this heuristic, we can iteratively assign the most powerful devices available in each iteration to the largest remaining stages. This method is the greedy algorithm we describe as follows:

Algorithm 1 Greedy Recovery Algorithm

```
1:  for $i \leftarrow m$ down to 1 do
2:    ▷ Iterate from the strongest GPU to the weakest GPU
3:    $num\_nodes \leftarrow n_i$
4:    $used\_device \leftarrow 0$
5:    $total\_device \leftarrow num\_nodes \times M$
6:    while $used\_device < total\_device$ do
7:      $min\_time \leftarrow \infty$
8:      $min\_idx \leftarrow -1$
9:      for $j \leftarrow 1$ to len($stages$) do
10:       $assign\_devices$
11:         $\leftarrow$ ceil(stages[$j$].device/$d_i$)
12:       $time \leftarrow$ try_assign($j, assign\_devices$)
13:       if $time < min\_time$ then
14:         $min\_time \leftarrow time$
15:         $min\_idx \leftarrow j$
16:       end if
17:     end for
18:     assign($min\_idx, assignments$)
19:     $used\_device \leftarrow used\_device + assignments$
20:   end while
21: end for
```

The greedy algorithm presents a viable solution for simpler configurations of the model and the cluster; however, it possesses inherent, critical limitations. Specifically, Oobleck is designed to generate a pipeline template in which no more than one node(M device) can be assigned per stage. This constraint can lead to substantial performance degradation, particularly when the difference in computational power between the weakest and the strongest devices is considerable. For example, as shown in figure6, assigning the A100 in any of the stages will lead to a significant reduce in that stage's execution time; however, the overall execution time will not change because it is still bounded by the stage with the longest execution cost, namely the critical path. Consequently, this mismatch results in a suboptimal utilization of available resources, severely impacting overall system performance.

6.2 Bottom-Up DP Algorithm

To address the inefficiencies of the greedy algorithm, particularly in scenarios involving the under-utilization of powerful devices, there is a pressing need to devise a method for merging shorter stages as shown in figure7. By generating a virtual template where the length of stages is $n * M$, it becomes feasible to merge any number of consecutive stages into a single, larger stage because each stage is assigned by one device. This approach ensures the full utilization of a powerful real device. This problem can be effectively modeled as a dynamic programming challenge akin to the coin-change problem, where the objective is to use all real devices to optimally recover from the generated virtual template, and minimize the total cost, primarily measured in terms of execution time.

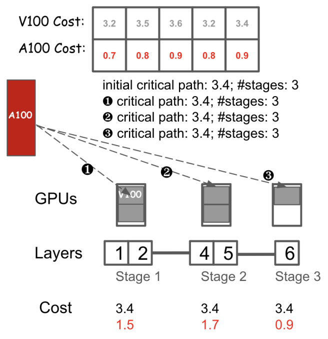

V100 Cost:
3.2
3.5
3.6
3.2
3.4
A100 Cost:
0.7
0.8
0.9
0.8
0.9
initial critical path: 3.4; #stages: 3
❶ critical path: 3.4; #stages: 3
❷ critical path: 3.4; #stages: 3
❸ critical path: 3.4; #stages: 3
A100
❶
❷
❸
GPUs
V100
Layers
1
2
4
5
6
Stage 1
Stage 2
Stage 3
Cost
3.4
1.5
3.4
1.7
3.4
0.9

Figure 6. Limitation of stage assignment: the overall execution time will still be bounded by the critical path, leading to a waste of device utilization

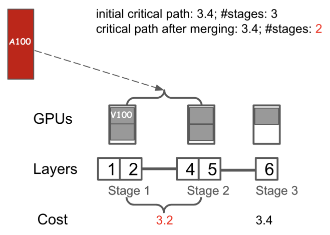

initial critical path: 3.4; #stages: 3
critical path after merging: 3.4; #stages: 2
A100
GPUs
V100
Layers
1
2
4
5
6
Stage 1
Stage 2
Stage 3
Cost
3.2
3.4

Figure 7. Motivation of stage merge: merging stage1 and stage2 can reduce the number of stages with the same critical path, thus reduce the overall iteration time

To solve the problem, we utilize a one-dimensional dynamic programming (DP) table. We define $dp[i]$ as a tuple $(\mathrm{d}, \mathrm{c}, \mathrm{t})$, referring to that template t containing the first i virtual stages has minimum cost c given assigned_devices d $(d_1, ..., d_m)$. The length of the DP table corresponds to the number of virtual stages, and each element is initially set as $(\text{none}, \infty, \text{nil})$. The final result is $dp[S]$, where S is the number of virtual stages.

To support state transition in this algorithm, we need to pre-generate a set of choices, (i, x, y). One choice indicates that we are able to use x devices with type i to merge y virtual stages. Here we can make the assumption that the execution time of each virtual stage is uniform by the nature of Oobleck's pipeline generation algorithm. These xs and ys in the choice are derived based on the node folding devices d_i. For instance, if a powerful device is three times as powerful as the weakest device($d_i = 3$), then its choice might be to use one device to merge up to three consecutive virtual stages, or M devices to merge up to $3M$ consecutive virtual stages. For device type i, we can generate choice including $(i, 1, d_i), ..., (i, M, M * d_i)$

The update rule for the DP table is specified as follows by iterating on all possible choices. We use K to denote the cost of the template by merging stage $(i - choice.y)...i$ using $choice.x$ devices with type $choice.i$ to existing optimal template stored in $dp[i - choice.x].t$.

$$K = dp[i - \text{choice.y}].\text{c} \\ + cost(assign(\text{choice}, stages[i - \text{choice.y} : i])) \tag{6}$$

$$dp[i].c = \min(dp[i].\text{c}, K) \tag{7}$$

$$dp[i].d = \\ \begin{cases} dp[i].d & \text{if } dp[i].\text{cost} < K \\ dp[i - \text{choice.y}].d[choice.i] += choice.x & \text{else} \end{cases} \tag{8}$$

$$dp[i].t = \\ \begin{cases} dp[i].t & \text{if } dp[i].\text{cost} < K \\ merge(dp[i - \text{choice.y}].t + choice) & \text{else} \end{cases} \tag{9}$$

In high-level idea, we are choosing whether to apply the current choice on top of the existing optimal template $dp[i - choice.x]$ or not based on the overall execution cost. This DP framework allows for the flexible and efficient merging of stages to optimize device utilization.

In addition to the core algorithm, we have posed several constraints to ensure the correctness state transitions within the DP table. One critical constraint is to skip choices that would result in using more real devices than are available. Moreover, when merging a choice with previous states to assign to a new state, an infeasible previous state will propagate its in-feasibility to the subsequent state. This chain of in-feasibility ensures that only viable and efficient device-stage pairings are considered, optimizing resource utilization across the computational tasks. With these optimizations, our DP recovery algorithm is presented in Algorithm2:

Algorithm 2 Dynamic Programming Recovery Algorithm

```
Require: Generated $choices$ based on folding factors
Require: $dp[i] \leftarrow$ (infeasible, $+\infty, nil$) for all $i$
 1: for $i = 1$ to $S$ do #S is number of stages
 2:     for $j = 1$ in $m$ do #m is device types
 3:         for each $choice$ in $choices[j]$ do
 4:             $prev\_state \leftarrow dp[i - choice.stages]$
 5:             if $prev\_state$ is infeasible then
 6:                 continue
 7:             end if
 8:             $assigned\_device \leftarrow prev\_state.d$
 9:             $assigned\_device[choice.i] += choice.x$
10:             if $assigned\_device$ over limit then
11:                 continue
12:             end if
13:             $t \leftarrow$ merge_stage($prev\_state.t, choice$)
14:             $min\_cost \leftarrow$ min($dp[i].cost, t.cost$)
15:             if $min\_cost$ is better then
16:                 $dp[i] \leftarrow assigned\_device, min\_cost, t$
17:             end if
18:         end for
19:     end for
20: end for
21: return $dp[S]$
```

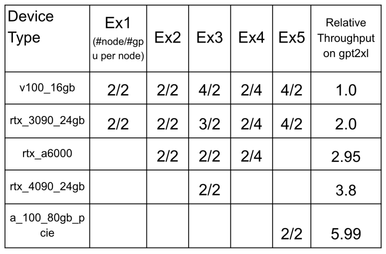
6.3 Time Complexity Analysis of the DP algorithm

Our DP algorithm iterates over all possible choices for each type of device as well as all the stages in the longest pipeline template that Oobleck generates. Thus given k types of devices, d devices in a node, and n pipeline stages generated for virtual cluster, the time complexity of our DP algorithm would be $O(kdn)$.

6.4 Limitation of the DP algorithm

Our DP algorithm follows the heuristic of using x devices with type i to merge $x * d_i$ stages. However, this assumes each stage has uniform execution cost. Although it holds for most of the time by the nature of Oobleck's template generation algorithm, the stages' cost may have a large variation in some extreme cases. For example, considering a gpt2 model with 32 layers, consisting of encoding layers, attention layers and decoding layers. It's known that the attention layer has much larger cost than the decoding layers. When recovering the template with 32 stages with only one layer in each stage, it would waste the device utilization if we merge only merge x stages with decoding layers using one device with type i. A global optimal algorithm would merge more than x stages with decoding layers for maximizing device utilization.

Another limitation is that our approach needs the total number of layers L to be larger or equal to the total number of virtual devices generated from node folding algorithm, namely $L >= n * M$. This is because the DP algorithm requires a template with $n * M$ stages, and each stage must has one or more layers.



Device Type	Ex1 (#node/#gpu per node)	Ex2	Ex3	Ex4	Ex5	Relative Throughput on gpt2xl
v100_16gb	2/2	2/2	4/2	2/4	4/2	1.0
rtx_3090_24gb	2/2	2/2	3/2	2/4	4/2	2.0
rtx_a6000		2/2	2/2	2/4		2.95
rtx_4090_24gb			2/2			3.8
a_100_80gb_pcie					2/2	5.99

Figure 8. 5 configurations of heterogeneous cluster consisting of 5 types of GPUs

7 EVALUATION

Since we haven't fully extended Oobleck's runtime(section 4.2 in Oobleck) to instantiate pipeline templates containing different devices type, an end-to-end evaluation of Oobleck+Ditto compared against other systems supporting planning on heterogeneous cluster is not applicable for our current progress. Therefore, we provide an Ablation Study of the effectiveness of Ditto, including the node folding algorithm presented in section 5 and the dynamic programming recovery algorithm presented in section 6.2. We evaluated Ditto on 1.5B parameter gpt2xl(Radford et al., 2019) with 6 configurations of heterogeneous cluster against the optimal algorithm using exhaustive searching discussed in section 3.

7.1 Experimental Setup

The heterogeneous clusters configuration is shown in figure8. There are 5 distinct configurations, each containing different combination of devices. From $Ex1...Ex3$, we increase the types of devices as well as the number of nodes in the cluster. In $Ex4$, we increase the number of devices in each node from 2 to 4; in $Ex5$, we added a_100_80gb_pcie, which is around 6x faster than V100, into the configuration. We collected each layer's profile in gpt2xl on an rtx_4090_24gb using Oobleck's profiler. The methodology and metrics of collecting profile is discussed in section 5.1. We synthesized each layer's profile on the other device types using the throughput scaling factors from GPU benchmarks by lambda lab(Lambda, 2024). We then feed each layer's profile on each device type together with the cluster configuration to Ditto and the optimal algorithm by exhaustive search for evaluation.

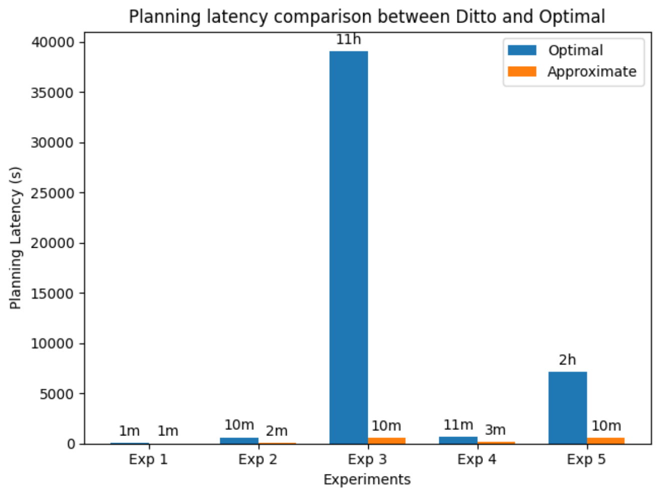


Figure 9. We compare the planning latency of Ditto(blue) against the optimal solution(orange). Y axis shows time. X axis corresponds to different heterogeneous cluster configurations described in 8

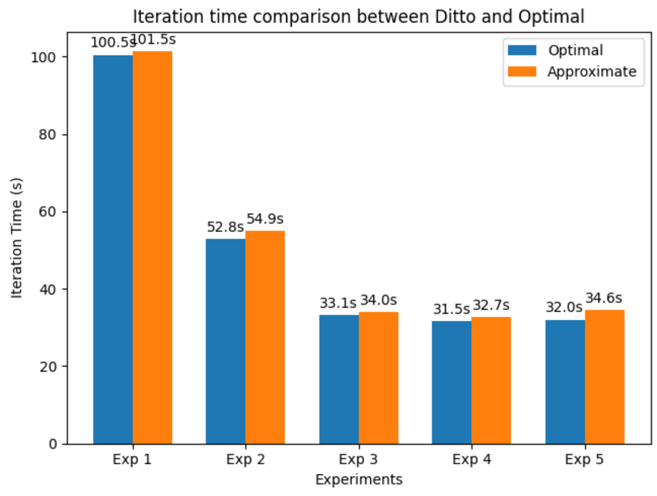


Figure 10. We compare the iteration time of pipeline template generated by Ditto(blue) with the optimal solution(orange). Y axis shows time. X axis corresponds to different heterogeneous cluster configurations described in figure 8

7.2 Experimental Results

The evaluation results of planning latency can be seen in figure 9. Ditto outperforms the optimal solution by orders of magnitude. In configuration $Ex3$ with 4 types of devices and 11 nodes in total, Ditto can generate the heterogeneous template in 10 minutes while the optimal solution needs to take 10 hours. In other smaller configurations, Ditto can generate the template in 5 minutes.

The evaluation results of the execution time T for the generated pipeline templates are shown in figure 10. T is calculated by the sum of pipeline drain time and pipeline steady state discussed in section3. In our experiments, the template generated by Ditto only has 8% overhead in maximum compared to the optimal solution. Given the savings in planning latency, we think that this slight performance drop is acceptable in this evaluation.

8 Related Work

Recent advancements in distributed training of large models have increasingly focused on addressing the challenges of scalability and fault tolerance. There are also several systems capable of partitioning large models on heterogeneous cluster automatically or through manual annotation. This section highlights for us how Ditto aligns with and diverges from those systems.

Large-Scale Model Training. There are numerous advances in scaling distributed training using different different levels of parallelism on homogeneous cluster(Wang et al., 2019), (Jia et al., 2019), (Cai et al., 2021), (Tarnawski et al., 2021), (Athlur et al., 2022), (Huang et al., 2019), (Li & Hoefler, 2021), (Tarnawski et al., 2021), (Zheng et al., 2022), (Jang et al., 2023). Ditto is capable of providing these systems an illusion of homogeneous cluster and reconstruct the real plan based on heterogeneous cluster configuration. This makes Ditto orthogonal to any of these.

Distributed training with fault tolerance. There are also several proposals in recent years attempting to support fault tolerance in distributed training of large models: Varuna(Athlur et al., 2022) reconfigures the combination of different levels of parallelism when failures happen; Bamboo (Thorpe et al., 2023) supports fault tolerance by redundant computation in pipeline parallelism without full restart; Oobleck (Jang et al., 2023) achieves fast failure recovery by reinstantiating pipeline(s) from the pre-generated pipeline templates without requiring a full restart from checkpoints. However, all of these work assume the underlying cluster is homogeneous. Ditto builds on top of Oobleck and empowers Oobleck to generate pipeline templates with heterogeneous GPU devices, which lays the fundation of supporting efficient failure recovery on heterogeneous cluster.

Planning on heterogeneous cluster. HetPipe(Park et al., 2020) supports planning large models on heterogeneous cluster by grouping heterogeneous GPUs into virtual worker. Each virtual worker holds a microbatch in a pipelined manner, and data-level parallelism is achieved by multiple workers. Whale(Jia et al., 2022), in similar way, hides the complexity of heterogeneity by grouping one or more physical devices into one logical device. Whale configures the parallel configurations by user's annotation and shifting the workload based on hardware configurations. Crius(Xue et al., 2024) presents a codesign of scheduling and parallelization for training large models on heterogeneous cluster. It proposes a novel scheduling granularity called $Cell$. By predetermining the number of stages and resources assigned to each $Cell$, the search space of parallelization would be shrunk into the product of tensor-level parallelism and data-level parallelism. Different from these systems, Ditto can automatically explore the search space of different levels of parallelism and generate pipeline templates containing heterogeneous GPUs statically in polynomial time. This will not create any overhead in Oobleck's runtime when instantiating those templates.

9 FUTURE WORK

Our evaluation only presents a ablation study on the effectiveness of node folding and DP recovery algorithm compared with the exhaustive searching algorithm presented in section3. We will extend Oobleck's runtime to instantiate these pipeline templates such that we can evaluate the full system against those related work(Park et al., 2020),(Jia et al., 2022),(Xue et al., 2024) using real test bed.

Our existing work does not consider the out of memory(OOM) issues on heterogeneous cluster for simplicity. To support this, we can either change the nold folding algorithm such that it will conservatively generate d_i based on the memory capacity or pose another constraints in the DP recovery algorithm to prevent merging stages on the devices that would lead to OOM issue.

We will extend Oobleck's full system to support fault tolerance on heterogeneous clusters by leveraging Ditto's ability to generate pipeline templates consisting of heterogeneous devices.

10 CONCLUSION

The research presented in this paper introduces Ditto, a novel framework designed to enhance the efficiency of distributed training on heterogeneous GPU clusters by automating the model partitioning using hybrid parallelism. The approach involves creating a virtual homogeneous environment to shrunk the parallelization space, and then accurately mapping the virtual pipeline template generated by existing planning system back onto the heterogeneous cluster using a dynamic programming approach in polynomial runtime. The ablation study on this approach indicates that Ditto is capable of generating pipeline template that has similar quality using an exhaustive search algorithm, but reduce the planning latency in orders of magnitude.

Further research could explore refining the cost models Ditto relies on, including better handling of communication costs and memory constraints, which could further improve the allocation efficiency and training speeds on complex, multi-node systems. As GPU architectures continue to evolve, tools like Ditto will be crucial for harnessing their full potential, ensuring that advancements in hardware can be effectively matched by software innovations in model training paradigms.

References

Torch-MLIR: The torch-mlir project aims to provide first class support from the PyTorch ecosystem to the MLIR ecosystem. `https://github.com/llvm/torch-mlir`, 2023. Accessed: 2024-04-25.

Athlur, S., Saran, N., Sivathanu, M., Ramjee, R., and Kwatra, N. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 472–487, 2022.

Cai, Z., Yan, X., Ma, K., Wu, Y., Huang, Y., Cheng, J., Su, T., and Yu, F. Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 33(8): 1967–1981, 2021.

Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., and Mace, J. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 443–462, 2020.

Hobbhahn, M. and Besiroglu, T. Trends in gpu price-performance, 2022. URL `https://epochai.org/blog/trends-in-gpu-price-performance`. Accessed: 2024-04-25.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Hwang, C., Kim, T., Kim, S., Shin, J., and Park, K. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pp. 721–739, 2021.

Jang, I., Yang, Z., Zhang, Z., Jin, X., and Chowdhury, M. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 382–395, 2023.

Jia, X., Jiang, L., Wang, A., Xiao, W., Shi, Z., Zhang, J., Li, X., Chen, L., Li, Y., Zheng, Z., et al. Whale: Efficient giant model training over heterogeneous {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 673–688, 2022.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

Jiang, J., Cui, B., Zhang, C., and Yu, L. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 463–478, 2017.

Lambda. GPU benchmarks for deep learning. `https://lambdalabs.com/gpu-benchmarks`, 2024. Accessed: 2024-04-23.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14. IEEE, 2021.

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

Park, J. H., Yun, G., Chang, M. Y., Nguyen, N. T., Lee, S., Choi, J., Noh, S. H., and Choi, Y.-r. {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)*, pp. 307–321, 2020.

Qi, H., Sparks, E. R., and Talwalkar, A. Paleo: A performance model for deep neural networks. In *International Conference on Learning Representations*, 2022.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI Blog*, 1 (8):9, 2019. URL `https://huggingface.co/openai-community/gpt2-xl`.

Sabne, A. Xla: Compiling machine learning for peak performance. *Google Res*, 2020.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Tarnawski, J. M., Narayanan, D., and Phanishayee, A. Piper: Multidimensional planner for dnn parallelization. *Advances in Neural Information Processing Systems*, 34: 24829–24840, 2021.

Thorpe, J., Zhao, P., Eyolfson, J., Qiao, Y., Jia, Z., Zhang, M., Netravali, R., and Xu, G. H. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 497–513, 2023.

Villalobos, P., Sevilla, J., Besiroglu, T., Heim, L., Ho, A., and Hobbhahn, M. Machine learning model sizes and the parameter gap. *arXiv preprint arXiv:2207.02852*, 2022.

Wang, M., Huang, C.-c., and Li, J. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–17, 2019.

Xue, C., Cui, W., Zhao, H., Chen, Q., Zhang, S., Yang, P., Yang, J., Li, S., and Guo, M. A codesign of scheduling and parallelization for large model training in heterogeneous clusters. *arXiv preprint arXiv:2403.16125*, 2024.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, 2022.