Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning

Lianmin Zheng, Zhuohan Li, and Hao Zhang, *UC Berkeley;* Yonghao Zhuang,
Shanghai Jiao Tong University; Zhifeng Chen and Yanping Huang, *Google;*
Yida Wang, *Amazon Web Services;* Yuanzhong Xu, *Google;* Danyang Zhuo,
Duke University; Eric P. Xing, *MBZUAI and Carnegie Mellon University;*
Joseph E. Gonzalez and Ion Stoica, *UC Berkeley*

This paper is included in the Proceedings of the
16th USENIX Symposium on Operating Systems
Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

Open access to the Proceedings of the
16th USENIX Symposium on Operating
Systems Design and Implementation
is sponsored by

∏ NetApp®

Alpa: Automating Inter- and Intra-Operator Parallelism
for Distributed Deep Learning

Lianmin Zheng[1,*]   Zhuohan Li[1,*]   Hao Zhang[1,*]   Yonghao Zhuang[4]
Zhifeng Chen[3]   Yanping Huang[3]   Yida Wang[2]   Yuanzhong Xu[3]   Danyang Zhuo[6]   Eric P. Xing[5]
Joseph E. Gonzalez[1]   Ion Stoica[1]

[1]*UC Berkeley* [2]*Amazon Web Services* [3]*Google* [4]*Shanghai Jiao Tong University*
[5]*MBZUAI, Carnegie Mellon University* [6]*Duke University*

Abstract

Alpa automates model-parallel training of large deep learning (DL) models by generating execution plans that unify data, operator, and pipeline parallelism. Existing model-parallel training systems either require users to manually create a parallelization plan or automatically generate one from a limited space of model parallelism configurations. They do not suffice to scale out complex DL models on distributed compute devices. Alpa distributes the training of large DL models by viewing parallelisms as two hierarchical levels: inter-operator and intra-operator parallelisms. Based on it, Alpa constructs a new hierarchical space for massive model-parallel execution plans. Alpa designs a number of compilation passes to automatically derive efficient parallel execution plans at each parallelism level. Alpa implements an efficient runtime to orchestrate the two-level parallel execution on distributed compute devices. Our evaluation shows Alpa generates parallelization plans that match or outperform hand-tuned model-parallel training systems even on models they are designed for. Unlike specialized systems, Alpa also generalizes to models with heterogeneous architectures and models without manually-designed plans. Alpa's source code is publicly available at https://github.com/alpa-projects/alpa.

1 Introduction

Several of the recent advances [10, 22, 49] in deep learning (DL) have been a direct result of significant increases in model size. For example, scaling language models, such as GPT-3, to hundreds of billions of parameters [10] and training on much larger datasets enabled fundamentally new capabilities.

However, training these extremely large models on distributed clusters currently requires a significant amount of engineering effort that is specific to both the model definition and the cluster environment. For example, training a large transformer-based language model requires heavy tuning and careful selection of multiple parallelism dimensions [40].

Training the large Mixture-of-Expert (MoE) transformers model [16, 31] on TPU clusters requires manually tuning the partitioning axis for each layer, whereas training the same model on an AWS GPU cluster calls for new pipeline schemes that can depend on the choices of partitioning (§8.1).

More generally, efficient large-scale model training requires tuning a complex combination of data, operator, and pipeline parallelization approaches at the granularity of the individual tensor operators. Correctly tuning the parallelization strategy has been shown [30, 33] to deliver an order of magnitude improvements in training performance, but depends on strong machine learning (ML) and system expertise.

Automating the parallelization of large-scale models would significantly accelerate ML research and production by enabling model developers to quickly explore new model designs without regard for the underlying system challenges. Unfortunately, it requires navigating a complex space of plans that grows exponentially with the dimensions of parallelism and the size of the model and cluster. For example, when all parallelism techniques are enabled, figuring out the execution plan involves answering a web of interdependent questions, such as how many data-parallel replicas to create, which axis to partition each operator along, how to split the model into pipeline stages, and how to map devices to the resulting parallel executables. The interplay of different parallelization methods and their strong dependence on model and cluster setups form a combinatorial space of plans to optimize. Recent efforts [17, 38, 55] to automatically parallelize model training are constrained to the space of a single model-parallelism approach, or rely on strong assumptions on the model and cluster specifications (§2.1).

Our key observation is that we can organize different parallelization techniques into a *hierarchical space* and map these parallelization techniques to the *hierarchical structure* of the compute cluster. Different parallelization techniques have different bandwidth requirements for communication, while a typical compute cluster has a corresponding structure: closely located devices can communicate with high bandwidth while distant devices have limited communication bandwidth.

---

*Lianmin, Zhuohan, and Hao contributed equally. Part of the work was done when Lianmin interned at Amazon and Zhuohan interned at Google.

将 ML 并行策略重新分类为 OP 内和 OP 间并行。OP 内并行沿一个或多个 tensor 轴划分 OP，并分布式地指派去设备；OP 间并行将模型切分为独立的 stages 并 pipeline。二者的区别在于划分粒度

通过指定每类并行策略的计划，可以将并行执行计划多级表示，这有许多好处：1) intra–op 并行的硬件利用率更高，但会在每 iter 带来切片之间的更多通信，以及划分 op 后的合并；2) inter–op 并行仅在相邻 stages 间通信，若适当切片则轻量，但会导致设备的 idle 时间。Alpa 将 intra–op 映射到高带宽设备，将 inter–op 映射到距离较远的低带宽设备，并分 level 优化（近似最优），全局非最优但性能较好（多级优化算法）

给定模型描述和集群配置，Alpa 编译器可以自动生成一系列编译 passes 来利用多级优化算法指定两级（intra & inter–op）并行的执行计划，包含一个新的 runtime 结构来在 devices meshes 间编排 inter–op 并行，并设计了许多系统优化来改善编译并处理跨 mesh 通信。注意，Alpa 在单个 mesh 内编排 intra–op 并行，在多 mesh 间编排 inter–op 并行。

With this observation in mind, in this paper, we take a different view from conventional data and model parallelisms, and re-categorize ML parallelization approaches as *intra-operator* and *inter-operator* parallelisms. Intra-operator parallelism partitions ML operators along one or more tensor axes (batch or non-batch) and dispatches the partitions to distributed devices (Fig. 1c); inter-operator parallelism, on the other hand, slices the model into disjoint stages and pipelines the execution of stages on different sets of devices (Fig. 1d). They take place at two different granularities of the model computation, differentiated by whether to partition operators.

Given that, a parallel execution plan can be expressed *hierarchically* by specifying the plan in each parallelism category, leading to a number of advantages. First, intra- and inter-operator parallelisms feature distinct characteristics: intra-operator parallelism has better device utilization, but results in communicating at every split and merge of partitioned operators, per training iteration; whereas inter-operator parallelism only communicates between adjacent stages, which can be light if sliced properly, but incurs device idle time due to scheduling constraints. We can harness the asymmetric nature of communication bandwidth in a compute cluster, and map intra-operator parallelism to devices connected with high communication bandwidth, while orchestrating the inter-operator parallelism between distant devices with relatively lower bandwidth in between. Second, this hierarchical design allows us to solve each level near-optimally as an individual tractable sub-problem. While the joint execution plan is not guaranteed globally optimal, they demonstrate strong performance empirically for training various large models.

Guided by this new problem formulation, we design and implement Alpa, the first compiler that automatically generates parallel execution plans covering all data, operator, and pipeline parallelisms. Given the model description and a cluster configuration, Alpa achieves this by partitioning the cluster into a number of *device meshes*, each of which contains devices with preferably high-bandwidth connections, and partitioning the computation graph of the model into *stages*. It assigns stages to device meshes, and automatically orchestrates intra-operator parallelisms on a device mesh and inter-operator parallelisms between device meshes.

In summary, we make the following contributions:
- We construct a two-level parallel execution plan space (Fig. 1e) where plans are specified hierarchically using inter- and intra-operator parallelisms.
- We design tractable optimization algorithms to derive near-optimal execution plans at each level.
- We implement Alpa, a compiler system for distributed DL on GPU clusters. Alpa features: (1) a set of compilation passes that generate execution plans using the hierarchical optimization algorithms, (2) a new runtime architecture that orchestrates the inter-op parallelism between device meshes, and (3) a number of system optimizations that improve compilation and address cross-mesh communication.
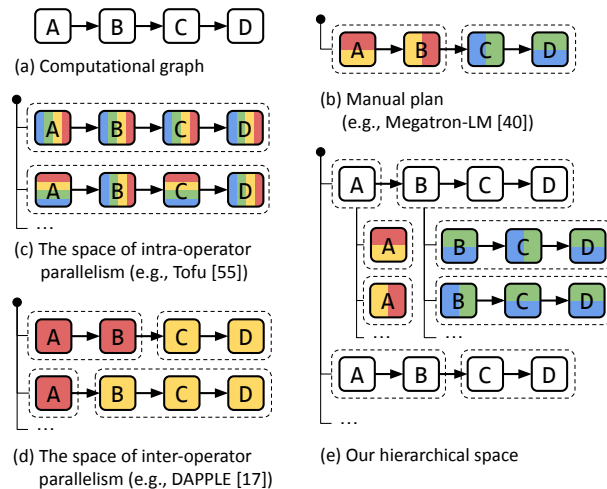

Figure 1: Generation of parallelization plans for a computational graph shown in (a). Different colors represent different devices, dashed boxes represent pipeline stages. (b) creates the plan manually. (c) and (d) automatically generate plans using only one of intra- and inter-operator parallelisms. (e) shows our approach that creates a hierarchical space to combine intra- and inter-operator parallelisms.

- We evaluate Alpa on training large models with billions of parameters. We compare Alpa with state-of-the-art distributed training systems on an Amazon EC2 cluster of 8 p3.16xlarge instances with 64 GPUs. On GPT [10] models, Alpa can match the specialized system Megatron-LM [40,49]. On GShard MoE models [31], compared to a hand-tuned system Deepspeed [45], Alpa achieves a 3.5× speedup on 2 nodes and a 9.7× speedup on 4 nodes. Unlike specialized systems, Alpa also generalizes to models without manual strategies and achieves an 80% linear scaling efficiency on Wide-ResNet [59] with 4 nodes. This means developers can get efficient model-parallel execution of large DL models out-of-the-box using Alpa.

2 Background: Distributed Deep Learning

DL computation is commonly represented by popular ML frameworks [1, 9, 42] as a dataflow graph. Edges in the graph represent multi-dimensional tensors; nodes are computational operators, such as matrix multiplication (matmul), that transform input tensors into output tensors. Training a DL model for one iteration consists of computing a loss by *forwarding* a batch of data through the graph, deriving the updates via a reverse *backward* pass, and applying the updates to the parameters via *weight update* operations. In practice, model developers define the dataflow graph. An execution engine then optimizes and executes it on a compute device.

When either the model or data is large that a single device cannot complete the training in a reasonable amount of time, we resort to ML parallelization approaches that parallelize the computation on distributed devices.
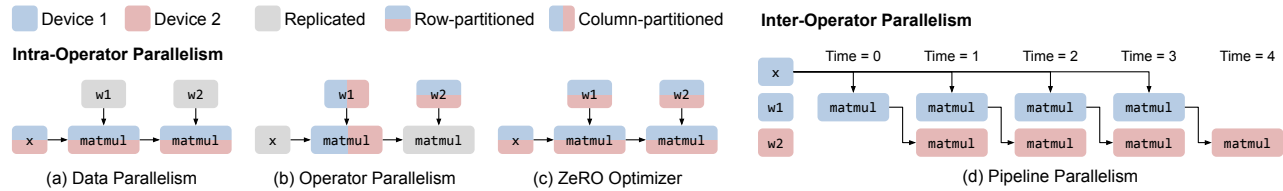
Figure 2: Common parallelization techniques for training a 2-layer Multi-layer Perceptron (MLP). Only the forward pass is shown. "x" is the input data. "w1" and "w2" are two weight matrices.

2.1 Conventional View of ML Parallelism

Existing ML parallelization approaches are typically categorized as data, operator, and pipeline parallelisms.

Data parallelism. In data parallelism, the training data is partitioned across distributed workers, but the model is replicated. Each worker computes the parameter updates on its independent data split, and synchronizes the updates with other workers before the weight update, so that all workers observe consistent model parameters throughout training.

Operator parallelism. When the model is too large to fit in one device, operator parallelism is an effective model parallelism option. Operator parallelism refers to approaches that partition the computation of a specific operator (abbreviated as *op* in the following text), such as matmul shown in Fig. 2b, along *non-batch* axes, and compute each part of the operator in parallel across multiple devices.

Because input tensors are jointly partitioned, when a device computes its op partition, the required portions of input tensors may not reside in its local memory. Communication is thus required to fetch the input data from other devices. When the tensors are partitioned evenly, i.e., SPMD [57], all devices will follow the same collective communication patterns such as all-reduce, all-gather, and all-to-all.

Pipeline parallelism. Instead of partitioning ops, pipeline parallelism places different groups of ops from the model graph, referred as *stages*, on different workers; meanwhile, it splits the training batch as a number of microbatches, and pipelines the forward and backward passes across microbatches on distributed workers, as Fig. 2d shows. Unlike operator parallelism, pipeline parallelism transfers intermediate activations at the forward and backward passes between different workers using point-to-point communication.

Manual combination of parallelisms. Recent development shows the approaches mentioned above need to be combined to scale out today's large DL models [40, 57]. The state-of-the-art training systems, such as Megatron-LM [40, 49], manually design a specialized execution plan that combines these parallelisms for transformer language models, which is also known as *3D Parallelism*. By assuming the model has the same transformer layer repeated, it assigns an equal number of layers to each pipeline stage and applies a hand-designed operator and data parallelism configuration uniformly for all layers. Despite the requirement of strong expertise, the manual plan cannot generalize to different models or different

cluster setups (§8.1).

Automatic combination of parallelisms. The configurations of each individual parallelism, their interdependence, and their dependence on model and cluster setups form an intractable space, which prevents the trivial realization of automatically combining these parallelisms. For examples, when coupled with operator parallelism, each time adding a data-parallel replica would require allocating a new set of devices (instead of one single device) as the worker, and figuring out the optimal operator parallelism configurations within those devices. When including pipeline parallelism, the optimal pipelining scheme depends on the data and operator parallelism choices of each pipeline stage and how devices are allocated for each stage. With this conventional view, prior explorations [17, 25, 55, 60] of auto-parallelization are limited to combining data parallelism with at most one model parallelism approach, which misses substantial performance opportunities. We next develop our view of ML parallelisms.

2.2 Intra- and Inter-Operator Parallelisms

Different from the conventional view, in this paper, we re-catalog existing parallelization approaches into two orthogonal categories: intra-operator and inter-operator parallelisms. They are distinguished by if they involve partitioning operators along any tensor axis. We next use the examples in Fig. 2 to introduce the two types of parallelisms.

Intra-operator parallelism. An operator works on multi-dimensional tensors. We can partition the tensor along some dimensions, assign the resulting partitioned computations to multiple devices, and let them execute different portions of the operator at the same time. We define all parallelization approaches using this workflow as intra-operator parallelism.

Fig. 2a-c illustrates the application of several typical instantiations of intra-op parallelism on an MLP. Data parallelism [29], by definition, belongs to intra-op parallelism – the input tensors and matmuls are partitioned along the batch dimension, and weight tensors are replicated. Alternatively, when the weights are very large, partitioning the weights (Fig. 2b) leads to the aforementioned operator parallelism adopted in Megatron-LM. Besides operators in the forward or backward passes, one can also partition the operators from the weight update phase, yielding the *weight update sharding* or equivalently the ZeRO [44, 56] technique, commonly comprehended as an optimization of data parallelism.

Due to the partitioning, collective communication is re-

quired at the split and merge of the operator. Hence, a key characteristic of intra-operator parallelism is that it results in substantial communication among distributed devices.

Inter-operator parallelism. We define inter-operator parallelism as the orthogonal class of approaches that *do not* perform operator partitioning, but instead, assign different operators of the graph to execute on distributed devices.

Fig. 2d illustrates the batch-splitting pipeline parallelism as a case of inter-operator parallelism.[2] The pipeline execution can follow different schedules, such as Gpipe [22], PipeDream [38], and synchronous 1F1B [17, 39]. We adopt the synchronous 1F1B schedule throughout this paper as it respects synchronous consistency, and has the same pipeline latency but lower peak memory usage compared to Gpipe.

In inter-operator parallelism, devices communicate only between pipeline stages, typically using point-to-point communication between device pairs. The required communication volume can be much less than the collective communication in intra-operator parallelism. Regardless of the schedule used, due to the data dependency between stages, inter-operator parallelism results in some devices being idle during the forward and backward computation.

By this categorization, the two parallelisms take place at different granularities of the DL computation and have distinct communication requirements, which happen to match the structure of today's compute clusters. We will leverage these properties to design hierarchical algorithms and compilation passes to auto-generate execution plans. Several concurrent work [2, 33, 39, 50] have proposed similar categorization, but Alpa is the first end-to-end system that uses this categorization to automatically generate parallel plans from the full space.

3 Overview

Alpa is a compiler that generates model-parallel execution plans by hierarchically optimizing the plan at two different levels: intra-op and inter-op parallelism. At the intra-op level, Alpa minimizes the cost of executing a stage (i.e., subgraph) of the computational graph, with respect to its intra-operator parallelism plan, on a given device mesh, which is a set of devices that may have high bandwidth between each other (e.g., GPUs within a single server). Different meshes might have different numbers of computing devices according to the workload assigned. At the inter-op level, Alpa minimizes the inter-op parallelization latency, with respect to how to slice the model and device cluster into stages and device meshes and how to map them as stage-mesh pairs. The inter-op optimization depends on knowing the execution cost of each stage-mesh pair reported by the intra-op optimizer. Through this hierarchical optimization process, Alpa generates the execution plan consisting of intra-op and inter-op plans which are

[2]Device placement [36] is another case of inter-op parallelism, which partitions the model graph and executes them on different devices but does not saturate pipelines using multiple microbatches. Hence pipeline parallelism is often seen as a better alternative to it because of less device idle time.
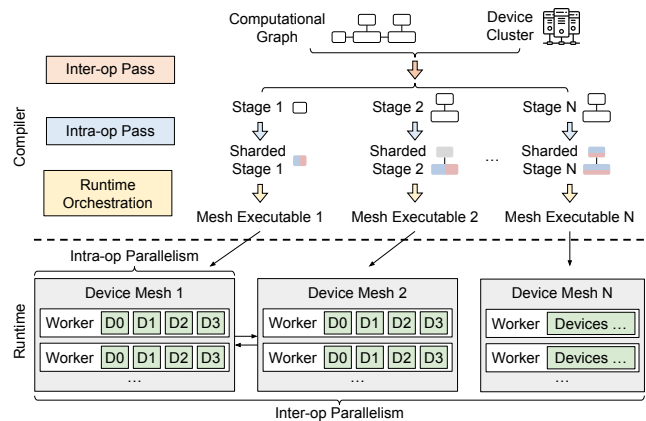

Figure 3: Compiler passes and runtime architecture. A sharded stage is a stage annotated with the sharding specs generated by intra-op pass.

```python
# Put @parallelize decorator on top of the Jax functions
@parallelize
def train_step(state, batch):
    def loss_func(params):
        out = state.forward(params, batch["x"])
        return jax.numpy.mean((out - batch["y"]) ** 2)

    grads = grad(loss_func)(state.params)
    new_state = state.apply_gradient(grads)
    return new_state

# A typical training loop
state = create_train_state()
for batch in data_loader:
    state = train_step(state, batch)
```

Figure 4: An example to demonstrate Alpa's API for Jax. The developers uses a Python decorator @parallelize to annotate functions that need to be parallelized. The rest of the program is kept intact.

locally near-optimal at their respective level of the hierarchy.

To achieve this, Alpa implements three novel compilation passes as Fig. 3 shows. Given a model description, in the form of a Jax [9] intermediate representation (IR), and a cluster configuration, the inter-op compilation pass slices the IR into a number of stages, and slices the device cluster into a number of device meshes. The inter-op pass uses a Dynamic Programming (DP) algorithm to assign stages to meshes and invokes the intra-op compilation pass on each stage-mesh pair, to query the execution cost of this assignment. Once invoked, the intra-op pass optimizes the intra-op parallel execution plan of the stage running on its assigned mesh, by minimizing its execution cost using an Integer Linear Programming (ILP) formulation, and reports the cost back to the inter-op pass. By repeatedly querying the intra-op pass for each allocation of a stage-mesh pair, the inter-op pass uses the DP to minimize the inter-op parallel execution latency and obtains the best slicing scheme of stages and meshes.

Given the output hierarchical plan and a designated pipeline-parallel schedule, each stage is first compiled as a

parallel executable on its located mesh. A runtime orchestration pass is invoked to fulfill the communication requirement between two adjacent stages that require communication between the two meshes they locate on. The runtime orchestration pass then generates static instructions specific to each mesh according to the pipeline-parallel schedule and invokes the execution on all meshes.

API. Alpa has a simple API shown in Fig. 4. Alpa requires developers to annotate functions to be parallelized, such as the `train_step()`, using a Python decorator `@parallelize`. Upon the first call to `train_step()`, Alpa traces the whole function to get the model IR, invokes the compilation, and converts the function to a parallel version.

Since the inter-op pass depends on the intra-op pass, in the following text, we first describe the intra-op pass, followed by the inter-op pass, and finally the runtime orchestration pass.

4 Intra-Operator Parallelism

Alpa optimizes the intra-operator parallelism plan within a device mesh. Alpa adopts the SPMD-style intra-op parallelism [31, 57] which partitions operators evenly across devices and executes the same instructions on all devices, as per the fact that devices within a single mesh have equivalent compute capability. This SPMD style significantly reduces the space of intra-op parallelism plans; meanwhile, it conveniently expresses and unifies many important approaches such as data parallelism, ZeRO, Megatron-LM's operator parallelism, and their combinations, which are not fully covered by existing automatic operators parallelism systems, such as Tofu [55] and FlexFlow [25]. Unlike systems that perform randomized search [25] or assume linear graphs [55], Alpa formalizes the problem as an integer linear programming (ILP) and shows it can be solved efficiently for computational graphs with tens of thousands of operators. Next, we describe the space of intra-op parallelism and our solution.

4.1 The Space of Intra-Operator Parallelism

Given an operator in the computational graph, there are multiple possible parallel algorithms to run it on a device mesh. For example, a matrix multiplication $C_{ij} = \sum_k A_{ik}B_{kj}$ corresponds to a three-level for-loop. To parallelize it, we can parallelize the loop i, loop j, loop k, or combinations of them across devices, which would have different computation and communication costs, require different layouts for the input tensors, and result in output tensors with different layouts. If an input tensor does not satisfy the layout requirement, a layout conversion is required, which introduces extra communication costs. The goal of the intra-op pass is to pick one parallel algorithm for every operator to minimize the execution time of the entire graph. Next, we formally define the device mesh and the layout of a tensor and discuss the cost of layout conversion.

Device mesh. A device mesh is a 2-dimensional logical view of a set of physical devices. Each device in the mesh has the same compute capability. Devices can communicate along

Table 1: Sharding specs of a 2-dimentional tensor on a 2×2 device mesh. A is a (N, M) tensor. The device mesh is [[Device 0, Device 1], [Device 2, Device 3]]. Each device stores a partition of A. The first column is the name of the sharding spec. The latter columns use Numpy syntax to describe the partitions stored on each device.

Spec	Device 0	Device 1	Device 2	Device 3
RR	$A[0:N,0:M]$	$A[0:N,0:M]$	$A[0:N,0:M]$	$A[0:N,0:M]$
S^0S^1	$A[0:\frac{N}{2},0:\frac{M}{2}]$	$A[0:\frac{N}{2},\frac{M}{2}:M]$	$A[\frac{N}{2}:N,0:\frac{M}{2}]$	$A[\frac{N}{2}:N,\frac{M}{2}:M]$
S^1S^0	$A[0:\frac{N}{2},0:\frac{M}{2}]$	$A[\frac{N}{2}:N,0:\frac{M}{2}]$	$A[0:\frac{N}{2},\frac{M}{2}:M]$	$A[\frac{N}{2}:N,\frac{M}{2}:M]$
S^0R	$A[0:\frac{N}{2},0:M]$	$A[0:\frac{N}{2},0:M]$	$A[\frac{N}{2}:N,0:M]$	$A[\frac{N}{2}:N,0:M]$
S^1R	$A[0:\frac{N}{2},0:M]$	$A[\frac{N}{2}:N,0:M]$	$A[0:\frac{N}{2},0:M]$	$A[\frac{N}{2}:N,0:M]$
RS^0	$A[0:N,0:\frac{M}{2}]$	$A[0:N,0:\frac{M}{2}]$	$A[0:N,\frac{M}{2}:M]$	$A[0:N,\frac{M}{2}:M]$
RS^1	$A[0:N,0:\frac{M}{2}]$	$A[0:N,\frac{M}{2}:M]$	$A[0:N,0:\frac{M}{2}]$	$A[0:N,\frac{M}{2}:M]$
$S^{01}R$	$A[0:\frac{N}{4},0:M]$	$A[\frac{N}{4}:\frac{N}{2},0:M]$	$A[\frac{N}{2}:\frac{3N}{4},0:M]$	$A[\frac{3N}{4}:N,0:M]$
RS^{01}	$A[0:N,0:\frac{M}{4}]$	$A[0:N,\frac{M}{4}:\frac{M}{2}]$	$A[0:N,\frac{M}{2}:\frac{3M}{4}]$	$A[0:N,\frac{3M}{4}:M]$

Table 2: Several cases of resharding. $all\text{-}gather(x,i)$ means an all-gather of x bytes along the i-th mesh axis. M is the size of the tensor. (n_0, n_1) is the mesh shape.

#	Src Spec	Dst Spec	Communication Cost
1	RR	S^0S^1	0
2	S^0R	RR	$all\text{-}gather(M,0)$
3	S^0S^1	S^0R	$all\text{-}gather(\frac{M}{n_0},1)$
4	S^0R	RS^0	$all\text{-}to\text{-}all(\frac{M}{n_0},0)$
5	S^0S^1	$S^{01}R$	$all\text{-}to\text{-}all(\frac{M}{n_0 \cdot n_1},1)$

the first mesh dimension and the second mesh dimension with different bandwidths. We assume different groups of devices along the same mesh dimension have the same communication performance. For a set of physical devices, there can be multiple logical views. For example, given 2 nodes and 8 GPUs per node (i.e., 16 devices in total), we can view them as a 2×8, 1×16, 4×4, 8×2, or 16×1 device mesh. The mapping between physical devices and the logical device mesh view is optimized by the inter-op pass (§5). In the rest of this section, we consider one fixed device mesh view.

Sharding Spec. We use *sharding spec* to define the layout of a tensor. For an *N-dimensional tensor*, its sharding spec is defined as $X_0X_1\cdots X_{n-1}$, where $X_i \in \{S,R\}$. If $X_i = S$, it means the i-th axis of the tensor is partitioned. Otherwise, the i-th axis is replicated. For example, for a 2-dimensional tensor (i.e., a matrix), SR means it is row-partitioned, RS means it is column-partitioned, SS means it is both row- and column-partitioned. RR means it is replicated without any partitioning. After we define which tensor axes are partitioned, we then have to map the partitioned tensor axes to mesh axes. We only consider 2-dimensional device meshes, so a partitioned tensor axis can be mapped to either the first or the second axis of the device mesh, or both. We added a superscript to S to denote the device assignment. For example, S^0 means

Table 3: Several parallel algorithms for a batched matmul $C_{b,i,j} = \sum_k A_{b,i,k} B_{b,k,j}$. The notation *all-reduce*(x,i) means an all-reduce of x bytes along the i-th mesh axis. M is the size of the output tensor. (n_0, n_1) is the mesh shape.

#	Parallel Mapping	Output Spec	Input Specs	Communication Cost
1	$i \to 0, j \to 1$	$RS^0 S^1$	$RS^0 R, RRS^1$	0
2	$i \to 0, k \to 1$	$RS^0 R$	$RS^0 S^1, RS^1 R$	*all-reduce*$(\frac{M}{n_0}, 1)$
3	$j \to 0, k \to 1$	RRS^0	$RRS^1, RS^1 S^0$	*all-reduce*$(\frac{M}{n_0}, 1)$
4	$b \to 0, i \to 1$	$S^0 S^1 R$	$S^0 S^1 R, S^0 RR$	0
5	$b \to 0, k \to 1$	$S^0 RR$	$S^0 RS^1, S^0 S^1 R$	*all-reduce*$(\frac{M}{n_0}, 1)$
6	$i \to \{0,1\}$	$RS^{01} R$	$RS^{01} R, RRR$	0
7	$k \to \{0,1\}$	RRR	$RRS^{01}, RS^{01} R$	*all-reduce*$(M, \{0,1\})$

the partitions are along the 0-th axis of the mesh, S^{01} means the partitions take place along both mesh axes. $S^0 R$ means the tensor is row-partitioned into two parts – The first part is replicated on device 0 and device 1, and the second part is replicated on device 2 and device 3. Table 1 shows all possible sharding specs of a 2-dimensional tensor on a 2×2 mesh with 4 devices.

Resharding. When an input tensor of an operator does not satisfy the sharding spec of the chosen parallel algorithm for the operator, a layout conversion, namely *resharding*, is required, which might require cross-device communication. Table 2 lists several cases of resharding. For instance, to convert a fully replicated tensor to any other sharding specs (case #1), we can slice the tensor locally without communication; to swap the partitioned axis (case #4), we perform an all-to-all.

Parallel algorithms of an operator. With the definitions above, consider parallelizing a batched matmul $C_{b,i,j} = \sum_k A_{b,i,k} B_{b,k,j}$ on a 2D mesh – Table 3 lists several intra-op parallel algorithms for a batched matmul. Algorithm#1 maps loop i to the 0-th mesh axis and loop j to the 1-th mesh axis, resulting in the output tensor C with a sharding spec $RS^0 S^1$. As the LHS operand $A_{b,i,k}$ and RHS operand $B_{b,k,j}$ both have only one parallelized index, their sharding specs are written as $RS^0 R$ and RRS^1, respectively. In this algorithm, each device has all its required input tiles (i.e., a partition of the tensor) stored locally to compute its output tile, so there is no communication cost. In Algorithm #2 in Table 3, when the reduction loop k is parallelized, all-reduce communication is needed to aggregate the partial sum. Similarly, we can derive the sharding specs and communication costs of other parallel algorithms for a batched matmul.

For other primitive operators such as convolution and reduction, we can get a list of possible parallel algorithms following a similar analysis of their math expressions. In the intra-op pass, the model graph is represented in XLA's HLO format [51], which summarizes common DL operators into less than 80 primitive operators, so we can manually enumerate the possible parallel algorithms for every primitive operator.

4.2 ILP Formulation

The total execution cost of a computational graph $G = (V,E)$ is the sum of the compute and communication costs on all nodes $v \in V$ and the resharding costs on all edges $e \in E$. We formulate the cost minimization as an ILP and solve it optimally with an off-the-shelf solver [18].

For node v, the number of possible parallel algorithms is k_v. It then has a communication cost vector c_v of length k_v, or $c_v \in \mathbb{R}^{k_v}$, where c_{vi} is the communication cost of the i-th algorithm. Similarly, node v has a compute cost vector $d_v \in \mathbb{R}^{k_v}$. For each node v, we define an one-hot decision vector $s_v \in \{0,1\}^{k_v}$ to represent the algorithm it uses. $s_{vi} = 1$ means we pick the i-th algorithm for node v. For the resharding cost between node v and node u, we define a resharding cost matrix $R_{vu} \in \mathbb{R}^{k_v \times k_u}$, where R_{vuij} is the resharding cost from the output of i-th strategy of node v to the input of j-th strategy of node u. The objective of the problem is

$$\min_s \sum_{v \in V} s_v^\mathsf{T}(c_v + d_v) + \sum_{(v,u) \in E} s_v^\mathsf{T} R_{vu} s_u, \qquad (1)$$

where the first term is the compute and communication cost of node v, and the second is the resharding cost of the edge (v,u). In this formulation, s is the variable, and the rest are constant values. The term $s_v^\mathsf{T} R_{vu} s_u$ in Eq. 1 is quadratic, and cannot be fed into an ILP solver. We linearize [19] the quadratic term by introducing a new decision vector $e_{vu} \in \{0,1\}^{k_v \cdot k_u}$ which represents the resharding decision between node v and u.

Although we can use profiling to get the accurate costs for c_v, d_v, and R_{vu}, we use the following methods to estimate them for simplicity. For communication costs d_v and R_{vu}, we compute the numbers of communicated bytes and divide them by the mesh dimension bandwidth to get the costs. For compute costs c_v, we set all of them as zero following the same motivation in [55]. This is reasonable because: (1) For heavy operators such as matmul, we do not allow replicated computation. All parallel algorithms always evenly divide the work to all devices, so all parallel algorithms of one operator have the same arithmetic complexity; (2) For lightweight operators such as element-wise operators, we allow replicated computation of them, but their computation costs are negligible.

To simplify the graph, we merge computationally-trivial operators, such as element-wise operators, transpose, and reduction, into one of their operands and propagate the sharding spec from the operand. This greatly reduces the number of nodes in the graph, thus the ILP problem size. We do a breath-first-search and compute the depth of each node. The node is merged to the deepest operand.

Once the parallel plan is decided by ILP, we also apply a set of post-ILP communication optimizations, such as replacing all-reduce with reduce-scatter and all-gather, whenever applicable, because the latter reduces the number of replicated tensors and corresponding computations, while keeping the communication volume the same. This achieves the effect of weight update sharding [56] or ZeRO optimizer [44].

当某个 OP 的输入 tensor 不符合该 OP 的 sharding spec 时，进行 layout conversion，即 resharding。主要分为 no communication、all-gather 和 all-to-all 三类，不同的 resharding cases 会带来不同的通信开销，原因是 devices 间用于参数传输的通信量和方式不同，注意 RR 时切分不需要通信，只要舍弃每个 device 上的部分参数

OP 的并行策略：对不同维度进行划分，其他维度进行 replicate。若划分后可以分别单独计算，每个设备有全部所需的输入 tiles，则无额外的通信开销；否则，需要用 all-reduce 来将 partial sum（没有完整的将 A 的一行和 B 的一列做矩阵乘，而是各自的一半）的结果聚合在一起

Intra-op pass 中，模型图被表示为 XLA 的 HLO 格式，将常见的 DL OPs 总结为 80 个不到的原语 OPs，所以可以人工枚举每个原语 OPs 的可能并行策略。

首先定义 Alpa 的 cost 模型：类似 Tofu，由于 Alpa 对大 OP 进行均匀切分，所有的并行方法有相同的算数复杂性；同时，小 OP 设备间存在的部分重复计算开销可忽略。因此，Alpa 仅关注通信开销和 reshard 开销，而非计算开销，对应 ILP 问题的目标函数是最小化模型图内所有节点的通信开销和所有边的 reshard 开销之和，这里除 one-hot 选择向量外，其他都为常量。对于 reshard 开销存在的二次项 sv^T * su，Alpa 引入新的决策向量（长度为 sv 和 su 长度之积）来降次，进而用已有的求解器进行优化

为了简化图，Alpa 将计算简单的 OP 合并，通过 BFS 确定深度最深的 OP，合并到该 OP。原因是推迟到最后的 depth 才计算，防止跨分支 merge 时出现依赖关系紊乱（例如分支 A 上已经执行到了，但分支 B 还没，此时执行 merged OP 显然不合理）

和 ZeRO 类似，将 all-reduce 拆分为 reduce-scatter 和 all-gather 后可以在 mesh 内共享优化器状态、梯度和参数，以节约内存

5 Inter-Operator Parallelism

In this section, we develop methods to slice the model and device cluster into stage-mesh pairs. Our optimization goal is to minimize the *end-to-end* pipeline execution latency for the entire computational graph. Previous works [17, 33] have considered simplified problems, such as assuming the device for each stage is pre-assigned, and all stages have fixed data or operator parallelism plan. Alpa rids these assumptions by jointly considering device mesh assignment and the existence of varying intra-op parallelism plans on each stage.

5.1 The Space for Inter-Operator Parallelism

Assume the computational graph contains a sequence of operators following the topology order of the graph[3], notated as $o_1, \ldots, o_K$, where the inputs of an operator o_k are from operators $o_1, \ldots, o_{k-1}$. We slice the operators into S stages $s_1, \ldots, s_S$, where each stage s_i consists of operators $(o_{l_i}, \ldots, o_{r_i})$, and we assign each stage s_i to a submesh of size $n_i \times m_i$, sliced from a computer cluster that contains devices, notated as the *cluster mesh* with shape $N \times M$. Let $t_i = t_{intra}(s_i, Mesh(n_i, m_i))$ be the latency of executing stage s_i on a submesh of $n_i \times m_i$, minimized by the ILP and reported back by the intra-op pass (§4). As visualized in Fig. 5, assuming we have B different input microbatches for the pipeline, the total minimum latency[4] for the entire computation graph is written as:

$$T^* = \min_{\substack{s_1, \ldots, s_S; \\ (n_1, m_1), \ldots, (n_S, m_S)}} \left\{ \sum_{i=1}^{S} t_i + (B-1) \cdot \max_{1 \le j \le S} \{t_j\} \right\}. \quad (2)$$

The overall latency contains two terms: the first term is the total latency of all stages, interpreted as the latency of the first microbatch going through the pipeline; the second term is the pipelined execution time for the rest of $B-1$ microbatches, which is bounded by the slowest stage (stage 3 in Fig. 5).

We aim to solve Eq. 2 with two additional constraints: (1) For an operator in the forward pass of the graph, we want to colocate it with its corresponded backward operator on the same submesh. Since backward propagation usually uses the similar set of tensors during forward propagation, this effectively reduces the amount of communication to fetch the required tensors generated at the forward pass to the backward pass. We use the sum of forward and backward latency for t_{intra}, so Eq. 2 reflects the total latency, including both forward and backward propagation. (2) We need the sliced submeshes $(n_1, m_1), \ldots, (n_S, m_S)$ to fully cover the $N \times M$ cluster mesh – we do not waste any compute device resources. We next elaborate on our DP formulation.

---

[3] We simply use the order of how users define each operator, reflected in the model IR, with the input operator as the origin. This allows us to leverage the inherent locality present in the user's program – closely related nodes in the graph will be more likely to be partitioned into the same stage.

[4] This formulation holds for GPipe and synchronous 1F1B schedules. Other pipeline schedules may require a different formulation.
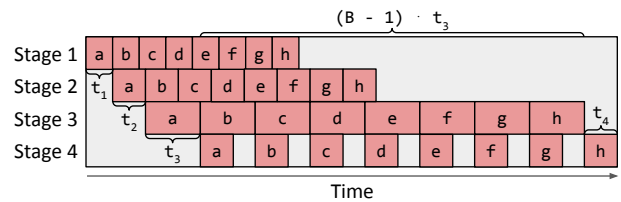

Figure 5: Illustration of the total latency of a pipeline, which is determined by two parts: the total latency of all stages ($t_1 + t_2 + t_3 + t_4$) and the latency of the slowest stage (($B-1$) · t_3).

5.2 DP Formulation

To ensure all submeshes $(n_1, m_1), \ldots, (n_S, m_S)$ fully cover the $N \times M$ cluster mesh, we reduce the available submesh shapes into two options: (1) one-dimensional submeshes of sizes $(1,1), (1,2), (1,4) \ldots (1,2^m)$ and (2) two-dimensional submeshes of size $(2,M), (3,M), \ldots, (N,M)$ that fully use the second dimension of the cluster mesh (i.e., on a GPU cluster, this means using all compute devices in each physical machine). We include a theorem in Appendix A that proves these submesh shapes can always fully cover the cluster mesh. To assign physical devices in the cluster to the resulting submeshes find by the DP algorithm, we enumerate by assigning devices to larger submeshes first and then to smaller ones. When there are multiple pipeline stages with the same submesh shape, we tend to put neighboring pipeline stages closer on the device mesh to reduce communication latency.

The simplification on submesh shapes works well for most available cloud deep learning setups: On AWS [3], the GPU instances have 1, 2, 4, or 8 GPUs; on GCP [20], the TPU instances have 8, 32, 128, 256 or 512 TPUs. The set of submesh shapes (n, m) excluded by the assumption is with $n > 1$ and $m < M$, which we observe lead to inferior results, since an alternative submesh with shape (n', M) where $n' \cdot M = n \cdot m$ has more devices that can communicate with high bandwidth. With this reduction, we only need to ensure that $\sum_{i=1}^{S} n_i \cdot m_i = N \cdot M$.

To find T^* in Eq. 2, we develop a DP algorithm. The DP first enumerates the second term $t_{max} = \max_{1 \le j \le S} t_j$ and minimizes the first term $t_{total}(t_{max}) = \sum_{1 \le i \le S} t_i$ for each different t_{max}. Specifically, we use the function $F(s, k, d; t_{max})$ to represent the minimal total latency when slicing operators o_k to o_K into s stages and putting them onto d devices so that the latency of each stage is less than t_{max}. We start with $F(0, K+1, 0; t_{max}) = 0$, and derive the optimal substructure of F as

$$F(s, k, d; t_{max}) \quad (3)$$
$$= \min_{\substack{k \le i \le K \\ n_s \cdot m_s \le d}} \left\{ \begin{array}{c} t_{intra}((o_k, \ldots, o_i), Mesh(n_s, m_s), s) \\ + F(s-1, i+1, d - n_s \cdot m_s; t_{max}) \\ | \ t_{intra}((o_k, \ldots, o_i), Mesh(n_s, m_s), s) \le t_{max} \end{array} \right\},$$

and derive the optimal total latency as

$$T^*(t_{max}) = \min_s \{F(s, 0, N \cdot M; t_{max})\} + (B-1) \cdot t_{max}. \quad (4)$$

The value of $t_{intra}((o_k, \ldots, o_i), Mesh(n_s, m_s), s)$ is determined by the intra-op pass. It is the lowest latency of executing the subgraph $(o_k, \ldots, o_i)$ on mesh $Mesh(n_s, m_s)$ with s subsequent stages. Note that $Mesh(n_s, m_s)$ is a set of physical devices – hence, we enumerate all the potential choices of logical device mesh shapes (n_l, m_l) satisfying $n_l \cdot m_l = n_s \cdot m_s$. For each choice, we query the intra-op pass with subgraph $(o_k, \ldots, o_i)$, logical mesh (n_l, m_l), and other intra-op options as inputs and get an intra-op plan. We then compile the subgraph with this plan and all other low-level compiler optimizations (e.g., fusion, memory planning) to get an executable for precise profiling. The executable is profiled in order to get the stage latency (t_l) and the memory required on each device to run the stage (mem_{stage}) and to store the intermediate activations (mem_{act}). We check whether the required memory fits the device memory (mem_{device}) according to the chosen pipeline execution schedule. For example, for 1F1B schedule [17, 39], we check

$$mem_{stage} + s \cdot mem_{act} \leq mem_{device}. \quad (5)$$

We pick the logical mesh shape that minimizes t_l and fits into the device memory. If none of them fits, we set $t_{intra} = \infty$.

Our algorithm builds on top of that in TeraPipe [33]. However, TeraPipe assumes all pipeline stages are the same, and the goal is to find the optimal way to batch input tokens into micro-batches of different sizes. Instead, Alpa aims to group the operators of a computational graph into different pipeline stages, while assuming the input micro-batches are of the same size. In addition, Alpa optimizes the mesh shape in the DP algorithm for each pipeline stage in inter-op parallelism. **Complexity.** Our DP algorithm computes the slicing in $O(K^3 NM(N + \log(M)))$ time for a fixed t_{max}. t_{max} has at most $O(K^2(N + \log(M)))$ choices: $t_{intra}((o_i, \ldots, o_j), Mesh(n_s, m_s))$ for $i, j = 1, \ldots, K$ and all the submesh choices. The complexity of this DP algorithm is thus $O(K^5 NM(N + \log(M))^2)$.

This complexity is not feasible for a large computational graph of more than ten thousand operators. To speed up this DP, we introduce a few practical optimizations.
Performance optimization #1: early pruning. We use one optimization that is similar to that in TeraPipe [33]. We enumerate t_{max} from small to large. When $B \cdot t_{max}$ is larger than the current best T^*, we immediately stop the enumeration. This is because larger t_{max} can no longer provide a better solution. Also, during enumeration of t_{max}, we only evaluate a choice of t_{max} if it is sufficiently larger than the last t_{max} (by at least ε). This allows the gap between the solution found by the DP algorithm and the global optima to be at most $B \cdot \varepsilon$. We empirically choose $\varepsilon = 10^{-6}$ s, and we find that the solution output by our algorithm is the same as the real optimal solution ($\varepsilon = 0$) for all our evaluated settings.

Algorithm 1 Inter-op pass summary.

1: **Input:** Model graph G and cluster C with shape (N, M).
2: **Output:** The minimal pipeline execution latency T^*.
3: // Preprocess graph.
4: $(o_1, \ldots, o_K) \leftarrow \text{Flatten}(G)$
5: $(l_1, \ldots, l_L) \leftarrow \text{OperatorClustering}(o_1, \ldots, o_K)$
6: // Run the intra-op pass to get costs of different stage-mesh pairs.
7: $submesh\_shapes \leftarrow \{(1,1),(1,2),(1,4),\ldots,(1,M)\} \cup \{(2,M),(3,M),\ldots,(N,M)\}$
8: **for** $1 \leq i \leq j \leq L$ **do**
9:   $stage \leftarrow (l_i, \ldots, l_j)$
10:   **for** $(n, m) \in submesh\_shapes$ **do**
11:     **for** s from 1 to L **do**
12:       $t\_intra(stage, Mesh(n, m), s) \leftarrow \infty$
13:     **end for**
14:     **for** $(n_l, m_l), opt \in \text{LogicalMeshShapeAndIntraOpOptions}(n, m)$ **do**
15:       $plan \leftarrow \text{IntraOpPass}(stage, Mesh(n_l, m_l), opt)$
16:       $t_l, mem_{stage}, mem_{act} \leftarrow \text{Profile}(plan)$
17:       **for** s satisfies Eq. 5 **do**
18:         **if** $t_l < t\_intra(stage, Mesh(n, m), s)$ **then**
19:           $t\_intra(stage, Mesh(n, m), s) \leftarrow t_l$
20:         **end if**
21:       **end for**
22:     **end for**
23:   **end for**
24: **end for**
25: // Run the inter-op dynamic programming
26: $T^* \leftarrow \infty$
27: **for** $t_{max} \in \text{SortedAndFilter}(t\_intra, \varepsilon)$ **do**
28:   **if** $B \cdot t_{max} \geq T^*$ **then**
29:     **break**
30:   **end if**
31:   $F(0, L+1, 0; t_{max}) \leftarrow 0$
32:   **for** s from 1 to L **do**
33:     **for** l from L down to 1 **do**
34:       **for** d from 1 to $N \cdot M$ **do**
35:         Compute $F(s, l, d; t_{max})$ according to Eq. 3
36:       **end for**
37:     **end for**
38:   **end for**
39:   $T^*(t_{max}) \leftarrow \min_s \{F(s, 0, N \cdot M; t_{max})\} + (B-1) \cdot t_{max}$
40:   **if** $T^*(t_{max}) < T^*$ **then**
41:     $T^* \leftarrow T^*(t_{max})$
42:   **end if**
43: **end for**

Performance optimization #2: operator clustering. Many operators in a computational graph are not computationally intensive (e.g., ReLU), and the exact placement of these operators has little impact on the total execution time. We develop another DP algorithm [4] to cluster neighboring operators to

reduce the total size of the graph Eq. 2 works on. We cluster the operators $(o_1, \ldots, o_K)$ into a series of layers[5] $(l_1, \ldots, l_L)$, where $L \ll K$. The goal of the algorithm is to merge two types of operators: (1) those that do not call for much computation but lengthen the computational graph and (2) neighboring operators that may cause substantial communication if put on different device meshes. We define function $G(k,r)$ as the minimum of maximal amount of data received by a single layer when clustering operators $(o_1, \ldots, o_k)$ into r layers. Note that G has the following optimal substructure:

$$G(k,r)$$

$$= \min_{1 \le i \le k} \left\{ \begin{array}{l} \max\{G(i-1, r-1), C(i,k)\} \\ FLOP(o_i, \ldots, o_k) \le \dfrac{(1+\delta)FLOP_{total}}{L} \end{array} \right\}, \quad (6)$$

where $C(i,k)$ denotes the total size of inputs of $(o_i, \ldots, o_k)$ received from $(o_1, \ldots, o_{i-1})$ and $FLOP_{total} = FLOP(o_1, \ldots, o_K)$ is the total FLOP of the whole computational graph. We make sure that each clustered layer's FLOP is within $1 + \delta$ times of the average FLOP per layer while minimizing the communication. For the solutions with the same communication cost, we choose the one with the most uniform structure by also minimizing the variance of per-layer FLOP. With our DP algorithm, we can compute the best layer clustering in $O(K^2 L)$ time. Note that L here is a hyperparameter to the algorithm. In practice, we choose a small L based on the number of devices and the number of heavy operators in the graph. We find different choices of L do not affect the final performance significantly.

Alg. 1 summarizes the workflow of the inter-op pass and illustrates its interactions with the intra-op pass in §4.

6 Parallelism Orchestration

After stages, device meshes, and their assignments are decided, at the intra-op level, Alpa compiles each stage against its assigned device mesh, respecting the intra-op parallelism plan output by the ILP solver. The compilation depends on XLA [51] and GSPMD [57], and generates parallel executables for each stage-mesh pair. When needed, the compilation automatically inserts collective communication primitives (see §4) to address the *within-mesh* communication caused by intra-op parallelism.

At the inter-op level, Alpa implements an additional parallelism orchestration pass to address the *cross-mesh* communication between stages, and generate static instructions for inter-op parallel execution.

Cross-mesh resharding. Existing manual systems, such as Megatron-LM [45, 49], constrain all pipeline stages to have the same degrees of data and tensor model parallelism, so the communication between pipeline stages is trivially realized by P2P send/recv between corresponded devices of

---

[5]Note that the clustering does not exactly reproduce the layers with original machine learning semantics in the model definition.
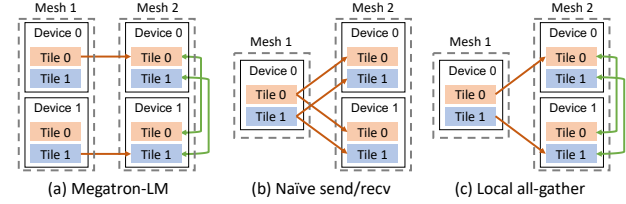

Figure 6: Cross-mesh resharding. Red arrows denote send/recv on slow connections. Green arrows denote all-gather on fast connections. (a) The scatter-gather optimization for equal mesh shapes in Megatron-LM. (b) The naive send/recv for unequal mesh shapes. (c) The generalized local all-gather optimization for unequal mesh shapes.

two equivalent device meshes (Fig. 6a). In Alpa, the device meshes holding two adjacent stages might have different mesh shapes, and the tensor to communicate between two stages might have different sharding specs (Fig. 6b and Fig. 6c). We call this communication pattern as *cross-mesh resharding*, which is a many-to-many multicast problem.

Given the sharding specs of the tensor on the sender and receiver mesh, Alpa generates a communication plan to address cross-mesh sharding in two iterations. In the first iteration, Alpa calculates the correspondences between tensor partitions (a.k.a. tiles) on the source and destination mesh. Based on that, it generates P2P send/recv primitives between the source devices and destination devices to fulfill the communication. It then takes a second iteration to identify opportunities where the destination tensor has a replication in its sharding spec. In this case, the tensor only needs to be transferred once between two meshes, then exchanged via all-gather across the devices on the destination mesh using its higher bandwidth (Fig. 6) – it rewrites send/recv generated at the first iteration into all-gather to avoid repeated communication.

We call this approach as *local all-gather* cross-mesh resharding. Since the communication between stages is normally small by our design, our experiments show that it performs satisfactorily well (§8.5). We defer the development of the optimal cross-mesh resharding plan to future work.

Generating pipeline execution instructions. As the final step, Alpa generates static execution instructions to launch the training on clusters. Since each stage has different sets of operators and may locate on meshes with different shapes, in contrast to many SPMD pipeline-parallel training systems [40, 57], Alpa adopts an MPMD-style runtime to orchestrate the inter-op parallel execution – Alpa generates distinct static execution instructions for each device mesh.

Alpa develops a set of instructions for inter-op parallel execution, including instructions for allocating and deallocating memory for tensors in a stage, communicating tensors between stages following the cross-mesh resharding plan, synchronization, and computation, etc. According to a user-selected pipeline schedule, Alpa uses a driver process to gen-

erate the instructions in advance and dispatches the whole instruction lists to each worker before execution, avoiding driver-worker coordination overheads during runtime.

7 Limitations and Discussion

In this section, we discuss advantages of our view of parallelisms and several limitations of our algorithms.

Compared to existing work that manually combines data, operator, and pipeline parallelism, such as 3D parallelism [45] and PTD-P [40], Alpa's hierarchical view of inter- and intra-op parallelisms significantly advances them with three major flexibility: (1) pipeline stages can contain an uneven number of operators or layers; (2) pipeline stages in Alpa might be mapped to device meshes with different shapes; (3) within each stage, the data and operator parallelism configuration is customized non-uniformly on an operator-by-operator basis. Together, they allow Alpa to unify all existing model parallelism approaches and generalize to model architectures and cluster setups with more heterogeneity.

Despite these advantages, Alpa's optimization algorithms currently have a few limitations:

• Alpa does not model the communication cost between different stages because the cross-stage communication cost is *by nature small*. In fact, modeling the cost in either the DP or ILP is possible, but would require enumerating exponentially more intra-op passes and DP states.

• The inter-op pass currently has a hyperparameter: the number of micro-batches *B*, which is not optimized by our current formulation but can be searched by enumeration.

• The inter-op pass models pipeline parallelism with a static linear schedule, without considering more dynamic schedules that, for example, parallelize different branches in a computational graph on different devices.

• Alpa does not optimize for the best scheme of overlapping computation and communication; Alpa can only handle static computational graphs with all tensor shapes known at compilation time.

Nevertheless, our results on weak scaling (§8) suggest that Alpa is able to generate near-optimal execution plans for many notable models.

8 Evaluation

Alpa is implemented using about 16K LoC in Python and 6K LoC in C++. Alpa uses Jax as the frontend and XLA as the backend. The compiler passes are implemented on Jax's and XLA's intermediate representation (i.e., Jaxpr and HLO). For the distributed runtime, we use Ray [37] actor to implement the device mesh worker, XLA runtime for executing computation, and NCCL [41] for communication.

We evaluate Alpa on training large-scale models with billions of parameters, including GPT-3 [10], GShard Mixture-of-Experts (MoE) [31], and Wide-ResNet [59]. The testbed is a typical cluster consisting of 8 nodes and 64 GPUs. Each node is an Amazon EC2 p3.16xlarge instance with 8 NVIDIA

Table 4: Models used in the end-to-end evaluation. LM = language model. IC = image classification.

Model	Task	Batch size	#params (billion)	Precision
GPT-3 [10]	LM	1024	0.35, 1.3, 2.6, 6.7, 15, 39	FP16
GShard MoE [31]	LM	1024	0.38, 1.3, 2.4, 10, 27, 70	FP16
Wide-ResNet [59]	IC	1536	0.25, 1.0, 2.0, 4.0, 6.7, 13	FP32

V100 16 GB GPUs, 64 vCPUs, and 488 GB memory. The 8 GPUs in a node are connected via NVLink. The 8 nodes are launched within one placement group with 25Gbps cross-node bandwidth.

We compare Alpa against two state-of-the-art distributed systems for training large-scale models on GPUs. We then isolate different compilation passes and perform ablation studies of our optimization algorithms. We also include a case study of the execution plans found by Alpa.

8.1 End-to-End Performance

Models and training workloads. We target three types of models listed in Table 4, covering models with both homogeneous and heterogeneous architectures. GPT-3 is a homogeneous transformer-based LM by stacking many transformer layers whose model parallelization plan has been extensively studied [40, 49]. GShard MoE is a mixed dense and sparse LM, where mixture-of-experts layers are used to replace the MLP at the end of a transformer, every two layers. Wide-ResNet is a variant of ResNet with larger channel sizes. It is vastly different from the transformer models and there are no existing manually designed strategies.

To study the ability to train large models, we follow common ML practice to scale the model size along with the number of GPUs, with the parameter range reported in Table 4. More precisely, for GPT-3, we increase the hidden size and the number of layers together with the number of GPUs following [40], whereas for MoE we mainly increase the number of experts suggested by [31, 57]. For Wide-ResNet, we increase the channel size and width factor in convolution layers. For each model, we adopt the suggested global batch size per ML practice [10, 31, 40, 59] to keep the same statistical behavior. We then tune the best microbatch size for each model and system configuration that maximizes the system performance. The gradients are accumulated across microbatches. The detailed model specifications are provided in Appendix B.

Baselines. For each model, we compare Alpa against a strong baseline. We use Megatron-LM v2 [40] as the baseline system for GPT-3. Megatron-LM is the state-of-the-art system for training homogeneous transformer-based LMs on GPUs. It combines data parallelism, pipeline parallelism, and manually-designed operator parallelism (denoted as TMP later). The combination of these techniques is controlled by three integer parameters that specify the parallelism degrees assigned to each technique. We grid-search the three parameters following the guidance of their paper and report the results of the best configuration. Megatron-LM is specialized for GPT-like models, so it does not support other models in Table 4.
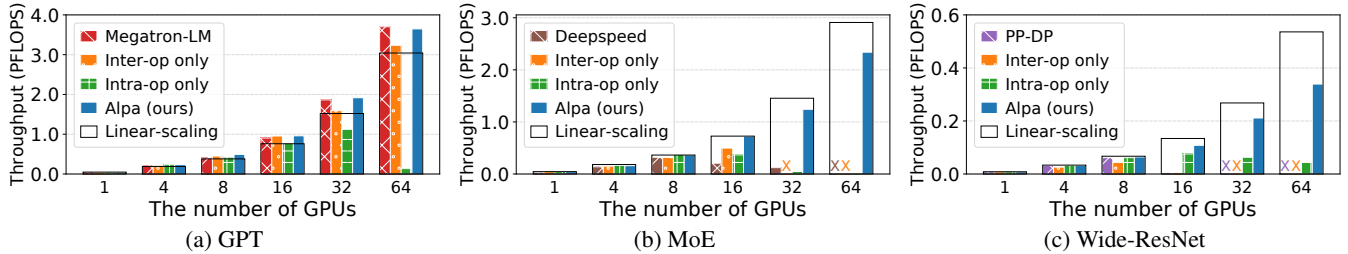
Figure 7: End-to-end evaluation results. "×" denotes out-of-memory. Black boxes represent linear scaling.

We use DeepSpeed [45] as the baseline for MoE. Deep-Speed provides a state-of-the-art implementation for training MoE on GPUs. It combines handcrafted operator parallelism for MoE layers and ZeRO-based [44] data parallelism. The combination of these techniques is controlled by several integer parameters that specify the parallelism degree assigned to each technique. We also grid-search them and report the best results. The performance of DeepSpeed on GPT-3 is similar to or worse than Megatron-LM, so we skip it on GPT-3. Note that original GShard-MoE [31] implementation is only available on TPUs, thus we do not include its results, though their strategies [31] are covered by Alpa 's strategy space.

For large Wide-ResNet, there is no specialized system or manually designed plan for it. We use Alpa to build a baseline "PP-DP" whose space only consists of data parallelism and pipeline parallelism, which mimics the parallelism space of PipeDream [38] and Dapple [17].

For all models, we also include the results of using Alpa with only one of intra- and inter-operator parallelism, which mimics the performance of some other auto-parallel systems. The open-source Flexflow [25] does not support the models we evaluate, as it lacks support for many necessary operators (e.g., layer normalization [5], mixed-precision operators). Tofu [55] only supports single node execution and is not open-sourced. Due to both theoretical and practical limitations, we do not include their results and we do not expect Flexflow or Tofu to outperform the state-of-the-art manual baselines in our evaluation.

Evaluation metrics. Alpa does not modify the semantics of the synchronous gradient descent algorithm, thus does not affect the model convergence. Therefore, we measure training throughput in our evaluation. We evaluate weak scaling of the system when increasing the model size along with the number of GPUs. Following [40], we use the aggregated peta floating-point operations per second (PFLOPS) of the whole cluster as the metric[6]. We measure it by running a few batches with dummy data after proper warmup. All our results (including those in later sections) have a standard deviation within 0.5%, so we skip the error bars in our figures.

GPT-3 results. The parallelization plan for GPT-3 has been extensively studied [10, 33, 40]. We observe in Fig. 7a that

this manual plan with the best grid-searched parameters enables Megatron-LM to achieve super-linear weak scaling on GPT-3. Nevertheless, compared to Megatron-LM, Alpa automatically generates execution plans and even achieves slightly better scaling on several settings. If compared to methods that only use intra-operator parallelism, our results are consistent with recent studies – "Intra-op only" performs poorly on >16 GPUs because even the best plan has to communicate tensors heavily on cross-node connections, making communication a bottleneck. Surprisingly, "Inter-op only" performs well and maintains linear scaling on up to 64 GPUs.

We investigate the grid-searched parameters of the manual plan on Megatron-LM, and compare it to the plan generated by Alpa. It reveals two major findings. First, in Megatron-LM, the best manual plan has TMP as 1, except in rare settings, such as fitting the 39B model on 64 GPUs, where pipeline parallelism alone is unable to fit the model (stage) in GPU memory; meanwhile, data parallelism is maximized whenever memory allows. In practice, gradient accumulation (GA) is turned on to achieve a desired global batch size (e.g., 1024 in our setting). GA amortizes the communication of data parallelism and reduces the bubbles of pipeline parallelism, but the communication of TMP grows linearly with GA steps, which puts TMP disadvantaged. Second, Alpa-generated plan closely resembles the best-performed ones in Megatron-LM, featuring (1) evenly-sized stages, (2) partitioning along the batch dimension in stages when memory is not stressed, but along non-batch dimensions when memory is stressed. One key difference between our plan and the manual plan is that Alpa also partitions the weight update operations when data parallelism exists, which contributes to the slight performance improvement over Megatron-LM. This attributes to the fact that Alpa, as a generic compiler system, can compose a wide range of parallelism approaches, while Megatron-LM, for now, misses weight update sharding support.

MoE results. DeepSpeed adopts a manual operator parallelism plan for MoE models, developed by GShard [31], called *expert parallelism*, which uses a simple rule: it partitions the expert axis for the operators in MoE layers, but switches back to data parallelism for non-expert layers. This expert parallelism is then combined with ZeRO data parallelism and TMP. All of these techniques belong to intra-operator parallelism. Unfortunately, DeepSpeed's specialized implementation does not include any inter-operator parallelism approach, which is
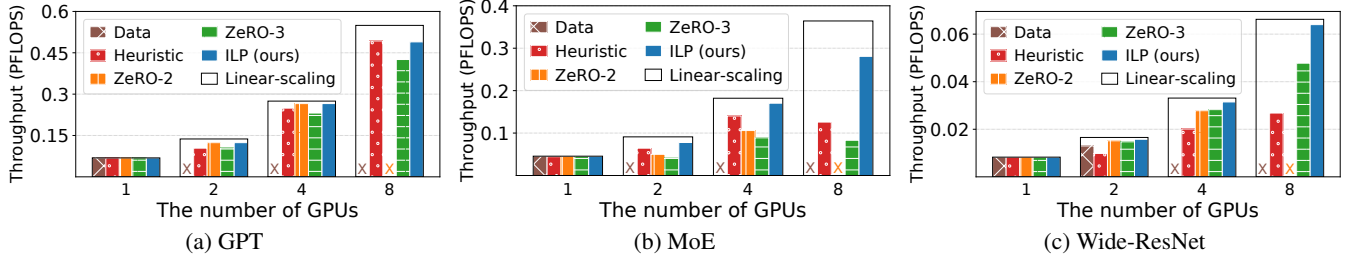
---

[6]As the models are different for different numbers of GPUs, we cannot measure scaling on the system throughput such as tokens per second or images per second.

Figure 8: Intra-operator parallelism ablation study. "×" denotes out-of-memory. Black boxes represent linear scaling.

required for scaling across multiple nodes with low inter-node bandwidth. Therefore, Deepspeed only maintains a good performance within a node ($\leq$ 8 GPUs) on this cluster. "Intra-op only" fails to scale across multiple nodes due to the same reason. "Inter-op only" runs out of memory on 32 GPUs and 64 GPUs because it is not easy to equally slice the model when the number of GPUs is larger than the number of layers of the model. The imbalanced slicing makes some memory-intensive stages run out of memory.

By contrast, Alpa automatically discovers the best execution plans that combine intra- and inter-operator parallelism. For intra-operator parallelism, Alpa finds a strategy similar to expert parallelism and combines it with ZeRO data parallelism, thanks to its ILP-based intra-op pass. Alpa then constructs stages and uses inter-operator parallelism to favor small communication volume on slow connections. Alpa maintains linear scaling on 16 GPUs and scales well to 64 GPUs. Compared to DeepSpeed, Alpa achieves $3.5\times$ speedup on 2 nodes and a $9.7\times$ speedup on 4 nodes.

Wide-ResNet results. Unlike the previous two models that stack the same layer, Wide-ResNet has a more heterogeneous architecture. As the data batch is forwarded through layers, the size of the activation tensor shrinks while the size of the weight tensor inflates. This leads to an imbalanced distribution of memory usage and compute intensity across layers. For this kind of model, it is difficult, if not impossible, to manually design a plan. However, Alpa still achieves a scalable performance on 32 GPUs with 80% scaling. The baselines "PP-DP" and "Inter-op only" run out of memory when training large models, because they cannot partition weights to reduce the memory usage, and it is difficult to construct memory-balanced stages for them. "Intra-only" requires a lot of communication on slow connections, so it cannot scale across multiple nodes. A case study on the generated plan for Wide-ResNet is in §8.6.

8.2 Intra-Op Parallelism Ablation Study

We study the effectiveness of our intra-operator parallelism optimization algorithm. We compare our ILP-based solution against alternatives such as ZeRO optimizer and rule-based partitioning strategies.

Experimental setup. We run a weak scaling benchmark in terms of model size similar to §8.1, but disable pipeline parallelism and gradient accumulation to control variables. The
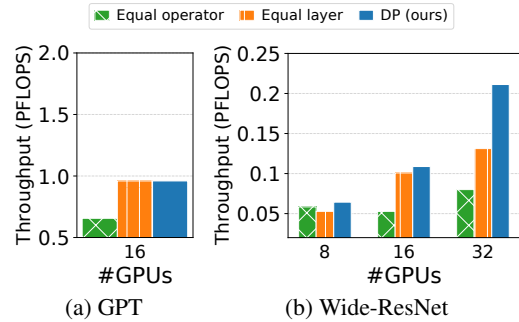

Figure 9: Inter-operator parallelism ablation study.

benchmark is done on one AWS p3.16xlarge instance with 8 GPUs. In order to simulate an execution environment of large-scale training in one node, we use larger hidden sizes, smaller batch sizes, and smaller numbers of layers, compared to the model configurations in §8.1.

Baselines. We compare automatic solutions for intra-operator parallelism. "Data" is vanilla data parallelism. "ZeRO-2" [44] is a memory-efficient version of data parallelism which partitions gradients and optimizer states. "ZeRO-3" [44] additionally partitions parameters on top of "ZeRO-2". "Heuristic" uses a rule combined with the sharding propagation in GSPMD. It marks the largest dimension of every input tensor as partitioned and runs sharding propagation to get the sharding specs for all nodes in the graph. "ILP" is our solution based on the ILP solver.

Results. As shown in Fig. 8, "Data" runs out of memory quickly and cannot train large models. "ZeRO-2" and "ZeRO-3" resolve the memory problem of data parallelism, but they do not optimize for communication as they always communicate the gradients. When the gradients are much larger than activations, their performance degenerates. "Heuristic" solves the memory issue by partitioning all tensors, but can be slowed down by larger communication. "Auto-sharding" performs best in all cases and maintains a near-linear scaling, because it figures out the correct partition plan that always minimizes the communication overhead.

8.3 Inter-Op Parallelism Ablation Study

We study the effectiveness of our inter-operator parallelism optimization algorithm. We use "DP" to denote our algorithm.
Experimental setup. We report the performance of three variants of our DP algorithm on GPT and Wide-ResNet. The
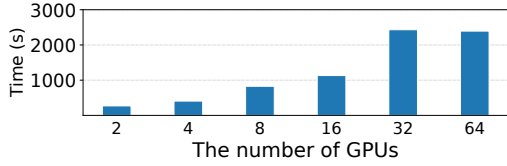
Figure 10: Alpa's compilation time on all GPT models. The model size and #GPUs are simultaneously scaled.

benchmark settings are the same as the settings in §8.1.

Baselines. We compare our DP algorithm with two rule-based algorithms. "Equal operator" disables our DP-based operator clustering but assigns the same number of operators to each cluster. "Equal layer" restricts our DP algorithm to use the same number of layers for all stages.

Results. Fig. 9 shows the result. "DP" always outperforms "Equal operator". This is because "Equal operator" merges operator that should be put onto different device meshes. Alpa's algorithm can cluster operators based on the communication cost and computation balance. Whether "DP" can outperform "Equal layer" depends on the model architecture. On homogeneous models like GPT, the solution of our DP algorithm uses the same number of layers for all stages, so "Equal layer" performs the same as "DP". On Wide-ResNet, the optimal solution can assign different layers to different stages, so "Equal layer" is worse than the full flexible DP algorithm. For Wide-ResNet on 32 GPUs, our algorithm outperforms "Equal operator" and "Equal layer" by 2.6× and 1.6×, respectively.

8.4 Compilation Time

Fig. 10 shows Alpa's compilation time for all the GPT settings in §8.1. The compilation time is a single run of the full Alg. 1 with a provided number of microbatches *B*. According to the result, Alpa scales to large models or large clusters well, because compilation time grows linearly with the size of the model and the number of GPUs in the cluster. Table 5 reports the compilation time breakdown for the largest GPT model in our evaluation (39B, 64 GPUs). Most of the time is spent on enumerating stage-mesh pairs and profiling them. For the compilation part, we accelerate it by compiling different stages in parallel with distributed workers. For profiling, we accelerate it using a simple cost model built at the XLA instruction level, which estimates the cost of matrix multiplication and communication primitives with a piece-wise linear model. With these optimizations, the compilation and search for a model take at most several hours, which is acceptable as it is much shorter than the actual training time, which can take several weeks.

8.5 Cross-Mesh Resharding

We evaluate our generalized local all-gather optimization for cross-mesh resharding between meshes with different shapes on Wide-ResNet, as shown in Fig. 11. "signal send/recv" is a synthetic case where we only send 1 signal byte between stages, which can be seen as the upper bound of the perfor-

Table 5: Compilation time breakdown of GPT-39B.

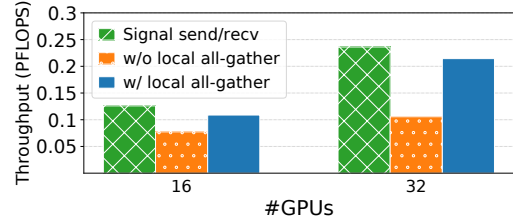| Steps	Ours	w/o optimization
Compilation	1582.66 s	> 16hr
Profiling	804.48 s	> 24hr
Stage Construction DP	1.65 s	N/A
Other	4.47 s	N/A
Total	2393.26 s	> 40hr

Figure 11: Cross-mesh resharding on Wide-ResNet.

mance. "w/o local all-gather" disables our local all-gather optimization and uses only send/recv. "w/ local all-gather" enables our local all-gather optimization to move more communication from slow connections to fast local connections, which brings 2.0× speedup on 32 GPUs.

8.6 Case Study: Wide-ResNet

We visualize the parallelization strategies Alpa finds for Wide-ResNet on 16 GPUs in Fig. 12. We also include the visualization of results on 4 and 8 GPUs in Appendix C. On 4 GPUs, Alpa uses only intra-operator parallelism. The intra-operator solution partitions along the batch axis for the first dozens of layers and then switches to partitioning the channel axis for the last few layers. On 16 GPUs, Alpa slices the model into 3 stages and assigns 4, 4, 8 GPUs to stage 1, 2, 3, respectively. Data parallelism is preferred in the first two stages because the activation tensors are larger than weight tensors. In the third stage, the ILP solver finds a non-trivial way of partitioning the convolution operators. The result shows that it can be opaque to manually create such a strategy for a heterogeneous model like Wide-ResNet, even for domain experts.

9 Related Work

Systems for data-parallel training. Horovod [47] and PyTorchDDP [32] are two commonly adopted data-parallel training systems that synchronize gradients using all-reduce. BytePS [26, 43] unifies all-reduce and parameter servers and utilizes heterogeneous resources in data center clusters. AutoDist [60] uses learning-based approaches to compose a data-parallel training strategy. ZeRO [44, 56] improves the memory usage of data parallelism by reducing replicated tensors. MiCS [61] minimizes the communication scale on top of ZeRO for better scalability on the public cloud. In Alpa, data parallelism [27] reduces to a special case of intra-operator parallelism – partitioned along the batch axis.

Systems for model-parallel training. The two major classes of model parallelisms have been discussed in §2. Mesh-
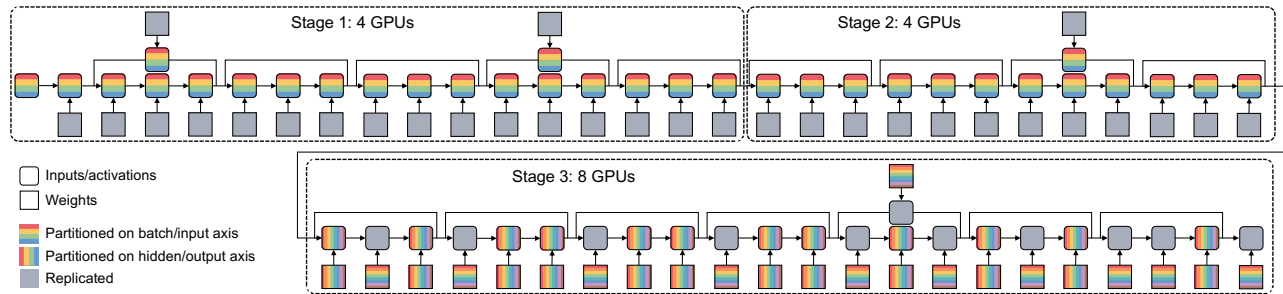
Figure 12: Visualization of the parallel strategy of Wide-ResNet on 16 GPUs. Different colors represent the devices a tensor is distributed on. Grey blocks indicate a tensor is replicated across the devices. The input data and resulting activation of each convolution and dense layer can be partitioned along the batch axis and the hidden axis. The weights can be partitioned along the input and output channel axis.

TensorFlow [48], GSPMD [31,57] and OneFlow [58] provide annotation APIs for users to manualy specify the intra-op parallel plan. ColocRL [36] puts disjoint model partitions on different devices *without pipelining*, thereby the concurrency happens only when there exist parallel branches in the model. In contrast, Gpipe [22] splits the input data into micro-batches and forms pipeline parallelisms. PipeDream [38, 39] improves GPipe by using asynchronous training algorithms, reducing memory usage, and integrating it with data parallelism. However, PipeDream is asynchronous while Alpa is a synchronous training system. TeraPipe [33] discovers a new pipeline parallelism dimension for transformer-based LMs. Google's Pathway system [7] is a concurrent work of Alpa. Pathway advocates a single controller runtime architecture combining "single program multiple data" (SPMD) and "multiple program multiple data" (MPMD) model. This is similar to Alpa's runtime part, where SPMD is used for intra-op parallelisms and MPMD is used for inter-op parallelism.

Automatic search for model-parallel plans. Another line of work focuses on the automatic discovery of model-parallel training plans. Tofu [55] develops a dynamic programming algorithm to generate the optimal intra-op strategy for *linear* graphs on *a single node*. FlexFlow [25] proposes a "SOAP" formulation and develops an MCMC-based randomized search algorithm. However, it only supports device placement without pipeline parallelism. Its search algorithm cannot scale to large graphs or clusters and does not have optimality guarantees. TensorOpt [11] develops a dynamic programming algorithm to automatically search for intra-op strategies that consider both memory and computation cost. Varuna [2] targets low-bandwidth clusters and focuses on automating pipeline and data parallelism. Piper [50] also finds a parallel strategy with both inter- and intra-op parallelism, but it relies on manually designed intra-op parallelism strategies and analyzes on a uniform network topology and asynchronous pipeline parallel schedules.

Techniques for training large-scale models. In addition to parallelization, there are other complementary techniques for training large-scale models, such as memory optimization [12,

14, 21, 23, 28, 46], communication compression [6, 53], and low-precision training [35]. Alpa can incorporate many of these techniques. For example, Alpa uses rematerialization to reduce memory usage and uses mixed-precision training to accelerate computation.

Compilers for deep learning. Compiler techniques have been introduced to optimize the execution of DL models [13,24,34,51,52,54,62]. Most of them focus on optimizing the computation for a single device. In contrast, Alpa is a compiler that supports a comprehensive space of execution plans for distributed training.

Distributed tensor computation in other domains. Besides deep learning, libraries and compilers for distributed tensor computation have been developed for linear algebra [8] and stencil computations [15]. Unlike Alpa, they do not consider necessary parallelization techniques for DL.

10   Conclusion

We present Alpa, a new architecture for automated model-parallel distributed training, built on top of a new view of machine learning parallelization approaches: intra- and inter-operator parallelisms. Alpa constructs a hierarchical space and uses a set of compilation passes to derive efficient parallel execution plans at each parallelism level. Alpa orchestrates the parallel execution on distributed compute devices on two different granularities. Coming up with an efficient parallelization plan for distributed model-parallel deep learning is historically a labor-intensive task, and we believe Alpa will democratize distributed model-parallel learning and accelerate the adoption of emerging large deep learning models.

11   Acknowledgement

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[3] AWS Cluster Configuratoins. https://aws.amazon.com/ec2/instance-types/p3/.

[4] Kevin Aydin, MohammadHossein Bateni, and Vahab Mirrokni. Distributed balanced partitioning via linear embedding. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, pages 387–396, 2016.

[5] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[6] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. Gradient compression supercharged high-performance data parallel dnn training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 359–375, 2021.

[7] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems*, 4, 2022.

[8] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. *ScaLAPACK users' guide*. SIAM, 1997.

[9] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[10] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[11] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1967–1981, 2021.

[12] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael W Mahoney, and Joseph E Gonzalez. Actnn: Reducing training memory footprint via 2-bit activation compressed training. In *International Conference on Machine Learning*, 2021.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[14] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[15] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. Distributed halide. *ACM SIGPLAN Notices*, 51(8):1–12, 2016.

[16] Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathy Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc V Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. Glam: Efficient scaling of language models with mixture-of-experts, 2021.

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

[18] John Forrest and Robin Lougee-Heimer. Cbc user guide. In *Emerging theory, methods, and applications*, pages 257–277. INFORMS, 2005.

[19] Richard J Forrester and Noah Hunt-Isaak. Computational comparison of exact solution methods for 0-1 quadratic programs: Recommendations for practitioners. *Journal of Applied Mathematics*, 2020, 2020.

[20] Google Clould TPU Cluster Configurations. https://cloud.google.com/tpu.

[21] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[22] 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*, 32:103–112, 2019.

[23] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *arXiv preprint arXiv:1910.02653*, 2019.

[24] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[25] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

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

[27] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[28] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616*, 2020.

[29] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[30] Woo-Yeon Lee, Yunseong Lee, Joo Seong Jeong, Gyeong-In Yu, Joo Yeon Kim, Ho Jin Park, Beomyeol Jeon, Wonwook Song, Gunhee Kim, Markus Weimer, et al. Automating system configuration of distributed machine learning. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 2057–2067. IEEE, 2019.

[31] 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. *arXiv preprint arXiv:2006.16668*, 2020.

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

[33] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. *arXiv preprint arXiv:2102.07988*, 2021.

[34] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897, 2020.

[35] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[36] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*, pages 2430–2439. PMLR, 2017.

[37] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging ai applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018.

[38] 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*, pages 1–15, 2019.

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

[40] 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*, pages 1–15, 2021.

[41] NVIDIA. The nvidia collective communication library, 2018.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: an imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[43] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[44] 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.

[45] 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*, pages 3505–3506, 2020.

[46] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. *arXiv preprint arXiv:2101.06840*, 2021.

[47] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[48] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for super-computers. *arXiv preprint arXiv:1811.02084*, 2018.

[49] 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.

[50] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. *Advances in Neural Information Processing Systems*, 34, 2021.

[51] Google XLA Team. Xla: Optimizing compiler for machine learning, 2017.

[52] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.

[53] Thijs Vogels, Sai Praneeth Karinireddy, and Martin Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. *Advances In Neural Information Processing Systems 32 (Nips 2019)*, 32(CONF), 2019.

[54] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. Pet: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54, 2021.

[55] 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*, pages 1–17, 2019.

[56] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang. Automatic cross-replica sharding of weight update in data-parallel training. *arXiv preprint arXiv:2004.13336*, 2020.

[57] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. Gspmd: General and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

[58] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, et al. Oneflow: Redesign the distributed

deep learning framework from scratch. *arXiv preprint arXiv:2110.15032*, 2021.

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

[60] Hao Zhang, Yuan Li, Zhijie Deng, Xiaodan Liang, Lawrence Carin, and Eric Xing. Autosync: Learning to synchronize for data-parallel distributed deep learning. *Advances in Neural Information Processing Systems*, 33, 2020.

[61] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. Mics: Near-linear scaling for training gigantic model on public cloud. *arXiv preprint arXiv:2205.00119*, 2022.

[62] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879, 2020.

A Proof of Submesh Shape Covering

We prove the following theorem which shows we can always find a solution that fully covers the cluster mesh (N, M) with our selected submesh shapes in §5.2: (1) one-dimensional submeshes of shape $(1, 1), (1, 2), (1, 4) \ldots (1, 2^m)$ where $2^m = M$ and (2) two-dimensional submeshes of shape $(2, M), (3, M), \ldots, (N, M)$.

Theorem 1. *For a list of submesh shapes $(n_1, m_1), \ldots (n_S, m_S)$, if $\sum_i n_i \cdot m_i = N \cdot M$ and each (n_i, m_i) satisfies either (1) $n_i = 1$ and $m_i = 2^{p_i}$ is a power of 2 or (2) $m_i = M$, then we can always cover the full (N, M) mesh where $M = 2^m$ with these submesh shapes.*

Proof. We start with putting the second type submesh into the full mesh. In this case, because $m_i = M$, these submeshes can cover the full second dimension of the full mesh. After putting all the second kind of submeshes into the mesh, we reduce the problem to fit a cluster mesh of shape (N, M) with submeshes with shape $(1, 2^{p_1}), \ldots, (1, 2^{p_S})$ where all $p_i \in \{0, 1, \ldots, m-1\}$. Note that now we have

$$2^{p_1} + \cdots + 2^{p_S} = N \cdot 2^m. \tag{7}$$

We start an induction on m. When $m = 1$, we have all $p_i = 0$ and thus all the submeshes are of shape $(1, 1)$, which means that all the submeshes can definitely cover the full mesh. Assume the above hold for all $m = 1, 2, \ldots, k-1$. When $m = k$, note that in this case the number of submeshes with $p_i = 0$ should be an even number, because otherwise the left hand side of Eq. 7 will be an odd number while the right hand side is always an even number. Then we can split all submeshes with shape $p_i = 0$ into pairs, and we co-locate each pair to form a $(1, 2)$ mesh. After this transformation, we have all $p_i > 0$, so we can subtract all p_i and m by 1 and reduce to $m = k-1$ case. Therefore, the theorem holds by induction. □

B Model Specifications

For GPT-3 models, we use sequence length = 1024 and vocabulary size = 51200 for all models. Other parameters of the models are listed in Table. 6. The last column is the number of GPUs used to train the corresponding model.

For GShard MoE models, we use sequence length = 1024 and vocabulary size = 32000 for all models. Other parameters of the models are listed in Table. 7. The last column is the number of GPUs used to train the corresponding model.

For Wide-ResNet models, we use input image size = (224, 224, 3) and #class = 1024 for all models. Other parameters of the models are listed in Table. 8. The last column is the number of GPUs used to train the corresponding model.

C Extra Case Study

We visualize the parallelization strategies Alpa finds for Wide-ResNet on 4 and 8 GPUs in Fig. 13.

Table 6: GPT-3 Model Specification

#params	Hidden size	#layers	#heads	#gpus
350M	1024	24	16	1
1.3B	2048	24	32	4
2.6B	2560	32	32	8
6.7B	4096	32	32	16
15B	5120	48	32	32
39B	8192	48	64	64

Table 7: GShard MoE Model Specification

#params	Hidden size	#layers	#heads	#experts	#gpus
380M	768	8	16	8	1
1.3B	768	16	16	16	4
2.4B	1024	16	16	16	8
10B	1536	16	16	32	16
27B	2048	16	32	48	32
70B	2048	32	32	64	64

Table 8: Wide-ResNet Model Specification

#params	#layers	Base channel	Width factor	#gpus
250M	50	160	2	1
1B	50	320	2	4
2B	50	448	2	8
4B	50	640	2	16
6.8B	50	320	16	32
13B	101	320	16	64
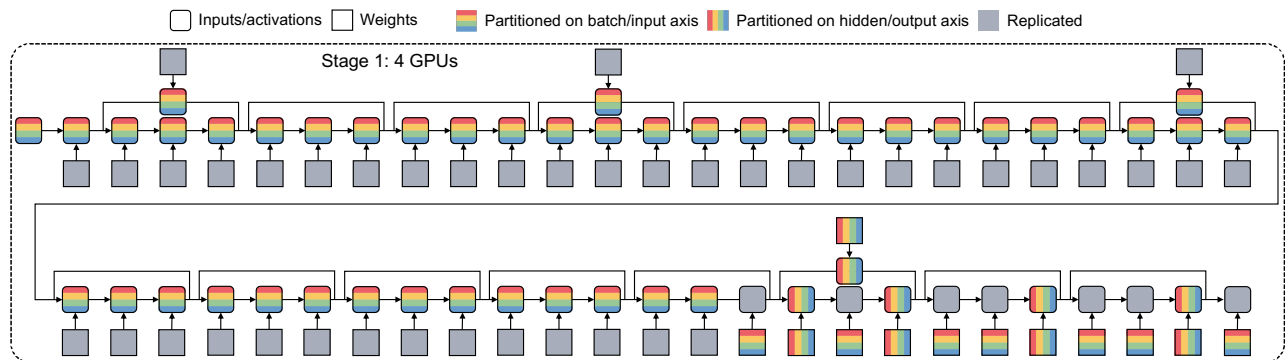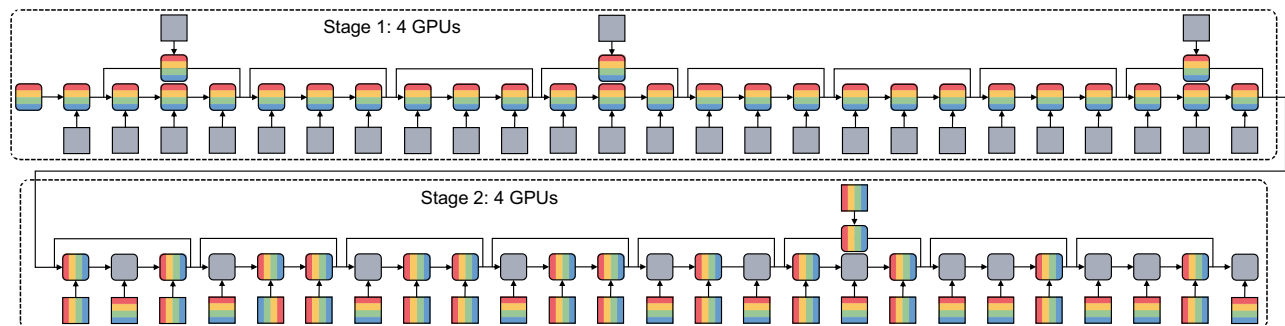
Figure 13: Visualization of the parallel strategy of Wide-ResNet on 4 and 8 GPUs. Different colors represent the devices a tensor is distributed on. Grey blocks indicate a tensor is replicated across all devices. The input data and resulting activation of each convolution or dense layer can be partitioned along the batch axis and the hidden axis. The weights can be partitioned along the input and output channel axis.