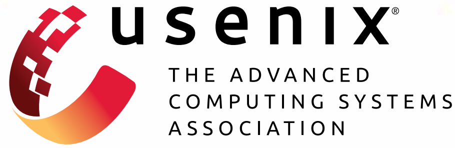


nnScaler: Constraint-Guided Parallelization Plan Generation for Deep Learning Training

Zhiqi Lin, *University of Science and Technology of China;* Youshan Miao, Quanlu Zhang, Fan Yang, and Yi Zhu, *Microsoft Research;* Cheng Li, *University of Science and Technology of China;* Saeed Maleki, *xAI;* Xu Cao, Ning Shang, Yilei Yang, Weijiang Xu, and Mao Yang, *Microsoft Research;* Lintao Zhang, *BaseBit Technologies;* Lidong Zhou, *Microsoft Research*

https://www.usenix.org/conference/osdi24/presentation/lin-zhiqi

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

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

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

nnScaler: Constraint-Guided Parallelization Plan Generation for Deep Learning Training

Zhiqi Lin[†*], Youshan Miao[‡], Quanlu Zhang[‡], Fan Yang[‡], Yi Zhu[‡], Cheng Li[†], Saeed Maleki[◊*], Xu Cao[‡], Ning Shang[‡], Yilei Yang[‡], Weijiang Xu[‡], Mao Yang[‡], Lintao Zhang[△*], Lidong Zhou[‡]
[†]*University of Science and Technology of China,*
[‡]*Microsoft Research,* [◊]*xAI,* [△]*BaseBit Technologies*

Abstract

With the growing model size of deep neural networks (DNN), deep learning training is increasingly relying on handcrafted search spaces to find efficient parallelization execution plans. However, our study shows that existing search spaces exclude plans that significantly impact the training performance of well-known DNN models (e.g., AlphaFold2) under important settings, such as when handling large embedding tables in large language models.

To address this problem, we propose nnScaler, a framework that generates efficient parallelization plans for deep learning training. Instead of relying on the existing search space, nnScaler advocates a more general approach that empowers domain experts to construct their own search space through three primitives, `op-trans`, `op-assign`, and `op-order`, which capture model transformation and the temporal-spatial scheduling of the transformed model of any parallelization plans. To avoid space explosion, nnScaler allows the application of *constraints* to those primitives during space construction. With the proposed primitives and constraints, nnScaler can compose existing search spaces as well as new ones. Experiments show that nnScaler can find new parallelization plans in new search spaces that achieve up to 3.5× speedup compared to solutions such as DeepSpeed, Megatron-LM, and Alpa for popular DNN models like Swin-Transformer and AlphaFold2.

1 Introduction

Deep neural networks (DNN) have shown remarkable success [2, 27, 35]. However, training a large DNN model today requires resources far exceeding the capacity of a single computing device, such as a GPU. Therefore, a common practice has been to partition a large model, schedule the partitioned model to a large number of GPUs, and then construct a well-coordinated execution plan across the GPUs (i.e., a parallelization plan) for deep learning training [24, 26, 39].

[*]This work was done when the authors were with Microsoft Research.

It is challenging to find an efficient parallelization plan for DNN model training. A DNN model is often represented as a data flow graph (DFG) that can consist of thousands of nodes [56], with each node representing a DNN operator, e.g., matrix multiplication. A parallelization plan requires deciding on a partitioning choice for each operator, which can have many different partition choices [26]. Additionally, each partitioning choice for all operators in the DFG further requires the selection of a spatial-temporal scheduling scheme from many scheduling options designed for thousands of GPUs. This creates a vast *search space* with prohibitive combinatorial complexity for identifying an effective parallelization plan that dictates model partitioning and scheduling.

Due to the immense search space, model training often depends on carefully designed parallelization plans. For example, Megatron-LM [50] incorporates the well-known, parameterized parallelization plans known as data/tensor/pipeline parallelism to support GPT-like models (§2). This approach essentially constructs a few well-studied classes of parallelization plans within the large search space. More recently, Alpa [65] organized parallelization plan choices into a two-level hierarchical space, where the system first searches a parallelization plan on pipeline (inter-operator) parallelism and then on tensor (intra-operator) parallelism within each pipeline stage. This approach offers a larger search space and so often results in better parallelization plans. However, existing search spaces exclude several configurations in the parallelization plans (§2, §4.2). Our study shows that this limitation significantly impacts training performance on well-known models such as Swin-Transformer [35] and AlphaFold2 [27] (§8).

While existing work studies specific parallelization plans or searches within a carefully-crafted search space, we argue that domain experts should be empowered with the capability to *compose their own search space*. Given the wide variety of model architectures and the expansive domain knowledge of an expert, this approach could expose more parallelization opportunities. To this end, we propose three primitives, `op-trans`, `op-assign`, and `op-order`, that enable the com-

position of search space with arbitrary model partitioning (`op-trans`), as well as spatial (`op-assign`) and/or temporal scheduling (`op-order`) of the partitioned model. We show that existing parallelization plans or search spaces can be elegantly expressed by the three primitives with *constraints* on model partitioning and spatial-temporal scheduling. More importantly, with the new constraints, searching within the space composed by the three primitives can lead to new parallelization plans that significantly outperform those found in existing search spaces or specific parallelization plans. Essentially, the three primitives, along with the necessary constraints, represent a more general abstraction to characterize parallelization plans.

Based on the above insight, we built nnScaler, a framework that facilitates the search, generation, and optimization of parallelization plans for deep learning training. Domain experts first use nnScaler to construct the desired search space for parallelization plans through the three primitives (§3). Specifically, given a model, `op-trans` designates how each operator can be partitioned; `op-assign` denotes the placement of each partitioned operator on GPUs; and `op-order` specifies the preferred temporal order across multiple operators when they are assigned to the same GPU.

nnScaler also allows the application of *constraints* to the primitives (§4). An example of a constraint applied to `op-trans` is one that only evenly splits an operator into 2, 4, 8, and 16 partitions. The use of constraints, especially those leveraging the characteristics of DNN models (e.g., the large embedding table in §4.2), greatly reduces the search space. As a result, with proper search policies applied to such a constrained search space (§5), nnScaler can discover unconventional parallelization plans that significantly outperform existing ones.

Given the sophisticated model partitioning and spatial-temporal scheduling enabled by nnScaler, a parallelization plan may deviate significantly from the original dataflow graph representing the DNN model. To ensure the correctness of a generated plan, nnScaler introduces vTensor-pTensor (§6), a tensor abstraction that tracks the "lineage" across operators before and after partitioning. This allows nnScaler to maintain correct data dependency during graph partitioning and detect cycles in the graph that could lead to deadlocks, thereby excluding invalid plans. Moreover, vTensor-pTensor also enables automatic communication adaptation when an operator is split and assigned across multiple devices. Finally, nnScaler lowers the discovered parallelization plan into executable code, enabling parallel deep learning training on each device.

Implemented based on PyTorch [43], nnScaler demonstrates great power and flexibility, facilitating the discovery of new parallelization plans (§4.2, §8) that achieve up to 3.5× speedup over existing parallel training systems, such as DeepSpeed [47], Megatron-LM [39], and Alpa [65], for popular deep learning models in computer vision (Swin-Transformer [34]), language translation (T5 [45]), and biological analysis (AlphaFold2 [27]). nnScaler has been used to develop, train, and finetune next generation deep learning models across Microsoft. The code is available in [5].

2 Background and Motivation

Search space for parallelization plans. A parallelization plan refers to a training execution plan that specifies the model partitioning and corresponding spatial-temporal scheduling scheme on a given set of GPUs. Training a large model with hundreds of billions of parameters requires thousands of GPUs [9]. A large model may consist of approximately 100 layers, each representing a sub-neural architecture (e.g., attention [58]) with tens of operators handling tensors with tens of thousands of dimension size (e.g., the hidden dimension). The vast partitioning choices (for a large model) and the enormous spatial-temporal scheduling choices (on a large number of GPUs) combine to create a prohibitively large, combinatorial search space for parallelization plans.

Existing approaches rely on well-studied, handcrafted parallelization plans or search space to address this problem. For example, *data parallelism*, a special parallelization plan, partitions an operator along the batch dimension of its associated tensors. These partitioned operators are then replicated across multiple devices (GPUs) and shared with the same model parameters (weights) to enable concurrent model training.

Tensor parallelism is a class of more general plans that permit the partitioning on dimensions not limited to the batch dimension [26, 50, 59]. This approach allows the partitioned operators to be distributed across different devices, accommodating models too large to fit into a single device.

As a large DNN model typically consists of multiple layers, it is also possible to partition a model into multiple stages, with each stage containing one or several layers. The stages are placed on different devices and executed in a pipeline, hence the name *pipeline parallelism*. To improve pipeline efficiency, a batch of training samples is further divided into micro-batches, and are then executed following a carefully designed temporal order [18, 24, 30].

The aforementioned parallelism schemes can be combined into a new scheme, known as *3D parallelism*, to further improve training efficiency. Megatron-LM [39] incorporates 3D parallelism, which integrates data, tensor, and pipeline parallelism in a parameterized manner to support GPT-like models. Given N devices, Megatron-LM partitions a model into K stages, with each stage divided into M partitions. The model is executed using K-stage pipeline parallelism and M-way tensor parallelism. For a sufficiently large N, Megatron-LM can also employ $(\frac{N}{M*K})$-way data parallelism to achieve further improvement in training performance. 3D parallelism represents a few well-studied classes of parallelization plans within the large search space.

Alpa [65] further generalizes these parallelism schemes to handcraft a two-level hierarchical search space. This hierarchy enables the use of efficient searching techniques like dynamic programming. Alpa has been shown to produce superior parallelization plans due to its larger search space, i.e., a combination of SPMD [61] (a generalized tensor-parallelism space) and pipeline parallelism.

Limitations of existing search space. Although existing handcrafted parallelization search space is shown effective for mainstream models with similar model architectures, it relies on assumptions that simplify the search and construction of parallelization plans. These simplifications, however, may exclude promising plans from considerations (§4.2).

In tensor parallelism, it is assumed that partitioned operators and their corresponding split tensors are distributed across *disjoint* devices. For example, to train a vision model with high fidelity images (e.g., [34]), tensor parallelism splits the large tensors associated with the large image and distributes the divided tensors among disjoint devices. This excludes cases where the split operators are placed on fewer devices, meaning multiple operators share one device and compute in a streamlined manner to reduce memory consumption and inter-device communication costs simultaneously [11] (detailed in §4.2 and §8).

Pipeline parallelism assumes that the training involves one forward pass and one backward pass. However, models like AlphaFold2 [17, 27] require three forward passes coupled with a single backward pass. This unconventional training approach renders existing pipeline parallelism [24, 38, 39] inapplicable.

Pipeline parallelism also assumes that different pipeline stages are spread across *disjoint* devices and prohibit any two stages from sharing the same set of devices through temporal multiplexing. For example, multi-lingual LLMs [45, 62, 64] often employ a large embedding table in the early computational stage in the model. This results in significant GPU memory consumption (>40%) but small computation utilization (<5%). Given the disjoint device assignments in pipeline parallelism (and also in tensor parallelism), the imbalance in hardware utilization is unavoidable.

The later handcrafted search spaces (e.g., [61, 65]) that combine tensor and pipeline parallelism (and others), inherit these assumptions and, therefore, suffer from the same limitations. This motivated us to design a more *flexible* method for space construction that can enable domain experts to find more effective training plans for their models (§3, §4, §5).

New challenges due to flexibility. Introducing a more flexible way to construct parallelization plan space brings new challenges. While existing frameworks like Megatron-LM [50], Alpa [65], and DeepSpeed [47] only implement a few well-studied partitioning, scheduling, and communications schemes that support parallelization plans in well-understood parallelization spaces, the new space could uncover new ways of operator partitioning, new operator scheduling with unconventional communication patterns. Furthermore, more flexible parallelization plans are less studied and hence could be error-prone. To address the above challenges, we designed a compiling process to detect and prevent potential errors in parallelization plans (e.g., cycles in a transformed DFG), and to generate the runtime code with efficient communication operations for the discovered parallelization plan (§6).

Primitives	**Usage**
`op-trans(op,algo,n)`	`algo` ∈ `op.algos()` `n` ∈ ℕ, natural numbers
`op-assign(op,d)`	`d` ∈ **D**, a set of devices
`op-order(op1,op2)`	`op1` executes before `op2`

Table 1: Primitives for parallelization space construction.

3 Parallelization Search Space Construction

A parallelization plan can be naturally expressed by the *model partitioning* and the *spatial-temporal* scheduling of the partitioned model. Correspondingly, nnScaler proposes three primitives, `op-trans`, `op-assign`, and `op-order` (summarized in Table 1), to capture the three aspects of a parallelization plan. Combined, the primitives can be used to compose any search space for a parallelization plan given arbitrary models and accelerator devices.

`op-trans`. `op-trans(op,algo,n)` transforms an operator `op` into `n` sub-operators according to a transformation algorithm `algo`, selected from the algorithm set corresponding to the type of `op`. For example, `matmul`$(A_{i,k}, B_{k,j})$, the matrix multiplication operator, can be partitioned into two `matmul` operators along dimension i of tensor A while replicating tensor B. In fact, most operators can be partitioned along a certain dimension (e.g., i or k in A or B) of the associated tensors and the computation of partitioned (sub) operators would remain the same as that of the original operators. Based on this observation, nnScaler implements the partitioning algorithms for the major operators in most DNN models. Domain experts can then reuse the desired algorithm via the `algos()` interface. nnScaler can also integrate custom transformation algorithms, such as those developed by domain experts, for any given operator. Note that the transformation algorithm can be more than just operator partitioning. For instance, an operator can be augmented by an additional recomputing operator or a memory-swapping operator to save memory [11, 23, 28, 41, 53]. In this paper, we use the term "transformation" and "partitioning" interchangeably.

`op-assign`. Given a set of devices **D** and an operator `op`, `op-assign(op,d)` denotes that `op` will be executed on the `d`-th device in **D**.

`op-order`. When non-dependent operators, e.g., `op1` and `op2`, are assigned to the same device, `op-order(op1,op2)` ensures that `op1` must execute before `op2`. Execution order for non-dependent operators can play a vital role in training performance. For example, in pipeline parallelism, an operator in a pipeline stage can be partitioned into multiple micro-batches along the batch dimension. We denote these (sub)operators as $\texttt{op.mb}_1$, $\texttt{op.mb}_2$, etc., where $\texttt{mb}_i$ designates the corresponding microbatch ID. The operators $\texttt{op.mb}_i$ can be executed in an arbitrary order with regard to $\texttt{op.mb}_j$ ($i \neq j$). Nonetheless, various research shows that once these operators are being orchestrated carefully in temporal dimension, it is possible to minimize the pipeline "bubble" [24, 54] to significantly improve training efficiency.

With the three primitives mentioned above, domain experts can write Python codes to compose arbitrary search spaces for parallelization plans given any DNN model. These codes are not necessarily tied to specific DNN models. Consequently, nnScaler separates the model codes from the codes related to search space and search policy. Note that to ease programming efforts, `op` in the primitives can represent a sub-graph, where the primitive applies to each of the operators in the sub-graph.

Due to the flexibility of the primitives and the scale of large DNN models, the constructed parallelization search space often contains hundreds and thousands of operators with combinatorial search complexity. To address this issue, nnScaler allows domain experts to impose *constraints* when applying those primitives. These constraints can significantly reduce the search space (§4), thereby enabling effective search methods (§5).

4 Applying Constraints in the Search Space

In nnScaler, constraints are expressed as parameterized arguments to the primitives in Table 1. When all arguments become specific values, the whole space is reduced to a concrete parallelization plan. Below, we illustrate how well-studied parallelization plans like data, tensor, and pipeline parallelism can be expressed by using the three primitives and constraints (§4.1). Several new constraints that lead to novel parallelization plans are discussed in §4.2.

4.1 Constraints for Existing Search Spaces

Constraints for data and tensor parallelism. Table 2 shows the primitives and the associated constraints for data and tensor parallelism. Both data parallelism and tensor parallelism partition an operator evenly into `n` partitions. The partition is performed along a certain dimension, depicted by `algo`, where each partitioned sub-operator is assigned to a distinct device for concurrent execution, i.e., constraints ② and ③ in Table 2. Note that data parallelism always partitions along the batch dimension, hence the selection of `algo` is more restricted compared to tensor parallelism.

Primitives	Constraints
① `sub-ops = op-trans(op,algo,n)`	$n = \lvert \mathbf{D} \rvert$
② $\texttt{op-assign(sub-op}_i\texttt{,d}_i\texttt{)}$	$d_i, d_j \in \mathbf{D}$,
③ $\texttt{op-assign(sub-op}_j\texttt{,d}_j\texttt{)}$	$d_i \neq d_j$

Table 2: Constraints for data and tensor parallelisms.

Constraints for pipeline parallelism. Given a device set **D**, pipeline parallelism divides a model G in to sub-graphs G_i ($0 \leq i < \lvert \mathbf{D} \rvert$), where i denotes the i-th pipeline stage. And those sub-graphs will be assigned in disjoint devices, shown in Table 3.

To minimize the bubble, pipeline parallelism divides a batch of samples into micro-batches. A sub-graph, denoted as $\texttt{(G}_i\texttt{,n)}$, operates on the the `n`-th micro-batch. We further denote a forward pass subgraph as $\texttt{fG}_i$ and a backward pass subgraph as $\texttt{bG}_i$, the constraints to schedule the well-known 1F1B [24] pipeline parallelism can be summarized in Table 4.

Primitives	Constraints
① $\texttt{op-assign(G}_i\texttt{, d}_i\texttt{)}$	$d_i, d_j \in \mathbf{D}$,
② $\texttt{op-assign(G}_j\texttt{, d}_j\texttt{)}$	$d_i \neq d_j$

Table 3: Constraints for dividing a model G into $\lvert \mathbf{D} \rvert$ stages.

Primitives	Constraints
① $\texttt{op-order((fG}_i\texttt{,m),(fG}_i\texttt{,n))}$	$m < n$
② $\texttt{op-order((bG}_i\texttt{,m),(bG}_i\texttt{,n))}$	
③ $\texttt{op-order((fG}_i\texttt{,m+}ofst\texttt{),(bG}_i\texttt{,m))}$	$ofst = \lvert \mathbf{D} \rvert - i$,
④ $\texttt{op-order((bG}_i\texttt{,m),(fG}_i\texttt{,m+}ofst\texttt{+1))}$	$m \geq 0$

Table 4: Constraints for 1F1B schedule.

As illustrated in Figure 1, Constraints ① and ② in Table 4 ensure that: in stage i, the execution order of micro-batches must be the same for both forward and backward passes. That is, given any two micro-batches m and n, where $m < n$, fG_m should be executed before fG_n (①). The same applies to bG_m and bG_n in the backward pass (②).

Constraints ③ and ④ in Table 4 specify the subtle scheduling order of 1F1B. They define $ofst$, the offset with respect to the current stage. The earlier the stage in the pipeline, the larger the offset. Therefore, given G_i, the backward pass of the earlier microbatch should be executed later w.r.t. the forward pass (③). And the forward pass of the later micro-batch should be executed in adjacency to the backward pass of the earlier micro-batch (④).

The hierarchical combination of tensor parallelism and pipeline parallelism forms the space of Alpa [65], where tensor parallelism is nested within each stage of pipeline parallelism. This can be constructed by replacing d_i in the pipeline constraints in Table 3 with a set of devices D_i for each stage.

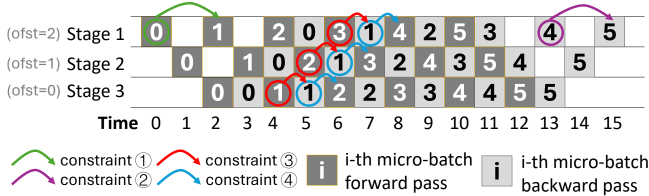


Figure 1: The constraints from Table 4 for 1F1B pipeline.

The stage is then applied with the constraints of tensor parallelism within D_i. For ease of exposition, the construction of this handcrafted parallelization space for certain sub-operators of a model, along with the constraints, is collectively defined in a general interface named `staged_spmd(ops, devices)`, to be used later.

4.2 New Constraints

In addition to existing search spaces, domain experts can apply new constraints to construct customized search spaces to search for new, more performant parallelization plans for various models, as we will elaborate next.

Constraints for Swin-Transformer. To enhance capability in vision tasks, there has been a growing trend to adopt higher resolution images to train large vision models such as Swin Transformer [34]. The use of larger images results in larger intermediate tensors during training, especially in the attention (Attn) and feedforward (FF) operators (for transformer-based models). It requires larger memory that a single GPU cannot accommodate.

Tensor parallelism is the standard practice used to address this issue. Given a pipeline, operators in Attn and FF are split and assigned to $|M_i|$ devices, where M_i denotes the set of devices accommodating operators in the i-th stage. Operators split by tensor parallelism are placed disjointedly, and so each device holds only one split operator. However, we observe that sometimes multiple split operators can share a single device and compute in a streamlined manner, resulting in fewer devices required for each pipeline stage and less memory consumption. Although the streamlined computing of multiple split operators may slow down the computing process, the reduced communications across fewer devices can lower cost and speed up the overall process.

Given any operator `op` from Attn and FF in stage i, let `sub_op` to be any transformed sub-operator of `op`. Suppose we allow C of such `sub_ops` to share one device, leading to a set of devices D_i assigned to stage i operators, where $|D_i| < |M_i|$. The constraints are as specified in Table 5. The rest operators can be described by the existing search space, namely the one defined in [65]. Note that C is a hyper-parameter where the value can be searched by the policy in §5.

Constraints for T5. Multi-lingual models such as T5 [45] often employ a large embedding table, say $\mathbf{E}$, which contains

Operators	Primitives	Constraints
`op` $\in$ {Attn $\cup$ FF}	`sub_ops = op-trans(op,algo,n)`	$n = C \cdot \mid \mathbf{D}_i \mid$
	`op-assign(sub_op`^j_i`,d`$_i$`)`	$0 \le j < \mid C \mid$ $d_i \in \mathbf{D}_i$

Table 5: Constraints for Swin-Transformer.

vocabulary embeddings from multiple languages [64]. The table $\mathbf{E}$, required only in the first and last layers of an LLM, incurs significant memory consumption but requires little computation cost. Pipeline parallelism would prioritize the device assignment to accommodate $\mathbf{E}$, leaving the remaining devices for the other operators. This arrangement results in imbalanced hardware utilization, with devices containing $\mathbf{E}$ exhibiting low GPU cycle usage but high memory usage.

Thanks to nnScaler's three primitives and constraints, we can split $\mathbf{E}$ across the entire device set $\mathbf{D}$. All other operators across all pipeline stages can then share the remaining resource left in $\mathbf{D}$ by constructing a search space following the conventional search space. These constraints, highlighted in Table 6, breaks the conventional assumption that operators in different pipeline stages cannot share the same set of devices. Similar solutions are also applicable to the training of graph neural networks [19].

Operators	Primitives	Constraints
`op` $\in \mathbf{E}$	`sub_ops = op-trans(op,algo,n)` `op-assign(sub_op`$_i$`,d`$_i$`)`	$n = \mid\mathbf{D}\mid$ $d_i \in \mathbf{D}$ $0 \le i < \mid D \mid$
`ops` $\notin \mathbf{E}$	`staged_spmd(ops,` $\mathbf{D}$`)`	

Table 6: Constraints for T5.

Constraints for AlphaFold2. In AlphaFold2 [27], training each micro-batch requires three forward passes and one backward pass, i.e., 3F1B. Traditional 1F1B pipeline parallelism cannot support this type of pattern. As shown on the left side of Figure 2, a naive approach of training one micro-batch after another is inefficient due to pipeline bubbles and the accumulation of many unnecessary intermediate results. Therefore, we decided to interleave the forward and backward passes across different micro-batches while maintaining constraints on temporal orders. Let f_pG_i denote the forward sub-graph fG_i at the i-th pipeline stage in the p-th forward pass, and let *ofst* be $S - i$, where S denotes the total number of pipeline stages. Table 7 highlights the constraints for 3F1B.

Constraints ① and ② in Table 7 interleave the three forward passes of consecutive micro-batches in decreasing order. Constraint ③ specifies that the smallest micro-batch in the last executed forward pass should be executed before the corresponding backward pass sub-graph on a micro-batch ID with an offset (*ofst*) relative to the current stage, where *ofst* is defined similarly to that in Figure 1 of §4.1.

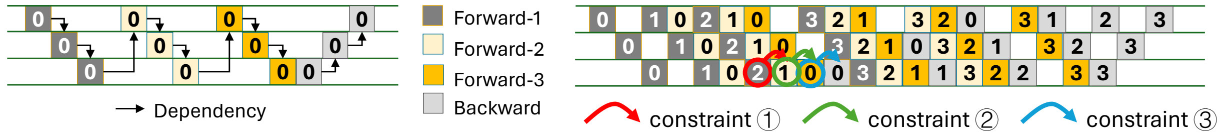


Figure 2: 3F1B schedule for AlphaFold2.

Primitives	Constraints
① `op-order((f1Gi,m+2),(f2Gi,m+1))`	$m \geq 0$
② `op-order((f2Gi,m+1),(f3Gi,m))`	$m > 0$
③ `op-order((f3Gi,m),(bGi,m-ofst))`	$m > ofst$

Table 7: Constraints for AlphaFold2.

In addition to Table 7, the search space for 3F1B also reuses the primitives and constraints in Tables 2 and 3. As shown on the right side of Figure 2, these constraints together form a space comprising unconventional parallelization plans that exhibit improved training performance (§8).

4.3 Discussion

Constraints are a powerful abstraction for customizing various parallelization plans and defining the search space for the plans. To design effective constraints, nnScaler assumes its users, usually domain experts, are knowledgeable on model architecture and parallel training. With such knowledge, it becomes intuitive to construct a search space using the three primitives. Based on our own experiences, effective constraints can be derived by identifying performance bottlenecks in the training, e.g., excessive GPU memory usage, computation/communication imbalance. The constraints can then be defined to alleviate the bottlenecks. And constraints can be refined iteratively along with the changing bottlenecks after the adjustment in constraints [33]. Through the refinement of constraints, nnScaler makes the generation of parallelization plans significantly easier than previous approaches.

5 Plan Search Policy

With the new user-defined search space, nnScaler incorporates a general policy framework to search for an efficient parallelization plan. As illustrated in Algorithm 1, the policy takes model graph G and a user-specified search space as inputs. We denote a space as C_{trans}, C_{assign}, C_{order}, corresponding to the three primitives `op-trans`, `op-assign` and `op-order`, along with augments associated with the constraints. The policy gradually shrinks the space with increasingly stringent constraints, ultimately reducing the space to a unique parallelization plan, denoted as C^{final}_{trans}, C^{final}_{assign}, C^{final}_{order}. A key feature of this policy framework is that it allows developers to "carve out" a sub-space from the new search space, where existing

Algorithm 1: The policy framework of plan search.

Input: G, Model graph; C_{trans}, C_{assign}, C_{order}, the space defined by the primitives with constraints.

Output: C^{final}_{trans}, C^{final}_{assign}, C^{final}_{order}, that determine a concrete parallelization plan.

`/* Operator partitioning & placement search */`
`/* Subgraph search with existing search algo */`

1 $G^{sub}, C^{sub}_{trans}, C^{sub}_{assign} \leftarrow$ GetSubSpace(G, C_{trans}, C_{assign});

2 $C^{new}_{trans}, C^{new}_{assign} \leftarrow$ Alpa($G^{sub}, C^{sub}_{trans}, C^{sub}_{assign}$);

`/* Search in the rest option space */`

3 $C_{trans}, C_{assign} \leftarrow$ ShrinkSpace($C_{trans}, C^{new}_{trans}, C_{assign}, C^{new}_{assign}$);

4 C^{final}_{trans} $C^{final}_{assign} \leftarrow$ ILP(G, C_{trans}, C_{assign}, objective=$eq.1$);

`/* Temporal ordering search */`

5 $C^{final}_{order} \leftarrow$ Tessel($G, C^{final}_{trans}, C^{final}_{assign}, C_{order}$);

6 **return** $C^{final}_{trans}, C^{final}_{assign}, C^{final}_{order}$; `/* a concrete plan */`

search polices are applicable. Specifically, the search process consists of two phases: operator partitioning and placement search, and temporal ordering search.

Operator Partitioning and Placement Search. The goal of this phase is to evenly distribute computations across devices while minimizing communication costs. Various partitioning options for an operator yield different communication costs. For instance, partitioning the batch dimension involves an allreduce on parameters, while partitioning parameters leads to replicating input activation tensors across devices. Different placement options for operators also result in varying execution times for each device. Therefore, the execution time on a device d is the sum of its assigned operators' computation time $Comp_d$ and the associated communication time $Comm_d$. Overall runtime is dictated by the slowest device [54, 65], which is formulated as:

$$\text{minimize} \quad \max_{d \in \mathbb{D}} \{Comp_d + Comm_d\}. \qquad (1)$$

By representing partitioning and placement options as integers, this optimization problem can be viewed as an integer linear programming problem, which is NP-hard.

With the application of constraints, the space in Equation 1 can be greatly reduced, thus enabling a faster search process (§8). nnScaler searches within a gradually reduced space by leveraging multiple policies. It firstly inspects the constructed search space and extracts a subspace (e.g., `staged_spmd`) that can leverage existing search policies like Alpa [65] through `GetSubSpace` (line 1 in Algorithm 1), an interface that re-

duces the search space through the input constraints. The extracted subspace may only consist of a subset of operators, leaving the rest operators undetermined. Once the transformation and placement decisions are made for the operator subset, the search space can be further reduced. Then, nnScaler fetches the reduced search space through `ShrinkSpace` (line 3 in Algorithm 1) and proceeds to use other policies (e.g., ILP solvers) within it, until finding the transformation and placement decisions for every operator.

For example, Table 5 reduces the operator assignment space to C operators per device; and Table 7 mainly specifies temporal order. The remaining subspace of these two cases can be organized like the space defined by `staged_spmd`. Additionally, Table 6 evenly pre-allocates the embedding table **E** to all devices evenly, with the remaining space corresponding exactly to `staged_spmd`. Consequently, the framework can apply the search policy in Alpa [65] to these sub-spaces to find a specific partitioning and assignment scheme for the involved operators (possibly a subset), i.e., C^{new}_{trans} and C^{new}_{assign} (line 2). These two new constraints, combined with their original versions, produce a smaller search space (line 3), where the framework can apply an ILP solver to find the final partitioning and assignment solution for the entire model, denoted as C^{final}_{trans} and C^{final}_{assign} (line 4). Note that as a general framework, users can replace the policies in Algorithm 1 by other search policies such as FlexFlow [26] or Tofu [59].

Temporal Ordering Search. After operator transformation and assignment, the temporal order of some operators is already specified by the data dependency in the transformed graph. However, it is possible for two operators on the same device to have no direct dependency, which means they can be executed in arbitrary orders. Moreover, for pipeline parallelism, the order of the same operator computed on different micro-batches within one batch is unspecified. nnScaler leverages Tessel [32], a state-of-the-art search policy, to determine the execution orders for these operators. Tessel groups operators within a micro-batch on each device into sub-graphs and formulates their execution order as an ILP problem. The optimization goal is to minimize the end-to-end execution latency of a mini-batch. Each sub-graph is assigned to an integer time slot, and the search, powered by Z3 Solver, enumerates possible order options without violating data dependencies. User-specified constraints on `op-order`, acting as Z3 constraints, play a crucial role in effectively reducing the search cost (line 5 in Algorithm 1).

Note that nnScaler does not claim contributions on individual search policies discussed in this section. It is the abstraction of primitives and constraints that makes the efficient search of parallelization plans possible.

6 Parallelization Plan Compilation

nnScaler compiles a model and the generated parallelization plan into executable codes, following the end-to-end process illustrated in Figure 3. The system first converts a deep learning model into a data flow graph, known as Graph IR. With the search space defined by the primitives and the associated constraints, nnScaler leverages a search policy to generate a parallelization plan. The plan compilation then applies the primitives and the constraints defined in the plan to the Graph IR. Data dependency tracking is performed during this step with the vTensor-pTensor abstraction. The resulting Graph IR, describing the new data dependency and the additional communication operations incurred due to operator distribution across devices, will be further materialized into parallel executable code.

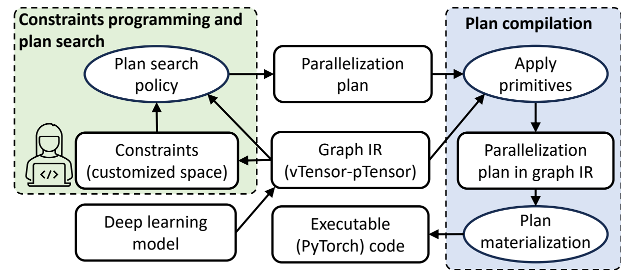


Figure 3: System overview of nnScaler.

Tensor Abstraction vTensor and pTensor are introduced to track changing data dependencies during the application of the three primitives. As depicted in Figure 4, a pTensor represents a tensor in the original logical model; vTensors are the resulting tensors after applying the three primitives to the pTensor. A vTensor links to a pTensor and maintains a mask indicating the accessed portion of the pTensor that this vTensor represents. A pTensor can be associated with multiple operators. At the top of Figure 4, the output of operator `A` serves as the input for operator `B`. Both operators are linked to the same pTensor through their respective vTensors.

With vTensor, each operator can be transformed, assigned, and ordered independently. When applying an `op-trans`, nnScaler partitions vTensors through the “mask”, leaving pTensors unchanged. For instance, in Figure 4, operator `A` only splits itself and its output vTensor, while the vTensor of operator `B` remains unaffected. For other type of primitives, vTensor’s mask remains unchanged. Therefore, given a producer vTensor (e.g., in `A`) and a consumer vTensor (e.g., in `B`) that are linked to the same pTensor, nnScaler can detect whether they have data dependency by intersecting their masks. With a dataflow graph, each operator in the graph consumes and produces vTensors according to underlying pTensors, thus facilitating the fine-grained data dependency tracking. During runtime execution, only vTensors will be instantiated to real GPU tensor instances.

With the data dependency tracking enabled by vTensor-pTensor abstraction during data flow graph transformation, nnScaler can detect cycles in the new graph that leads to

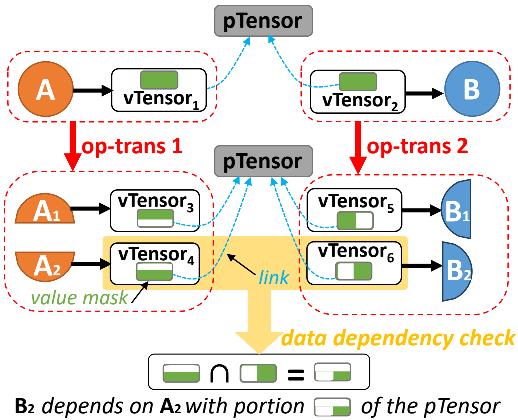


Figure 4: Tracks data dependencies with pTensor-vTensor.

deadlock, thus excluding invalid parallelization plans.

Data Dependency Materialization After applying primitives and constraints, nnScaler materializes the new data dependencies described by vTensor-pTensor into concrete data operations and communications. For a consumer vTensor (e.g., B_1 in Figure 4), nnScaler identifies dependent producer vTensors (e.g., A_1 and A_2) and inserts tensor manipulation operations, such as `torch.split` or `torch.chunk`, to extract the corresponding tensor fragments. When producers and consumers reside on different devices (due to `op-assign`), peer-to-peer send-recv communication operators [44] will be inserted during materialization.

To improve communication efficiency, certain communication patterns across vTensors within the same pTensor can be implemented using collective communication primitives, such as allgather, allreduce, or alltoall [44]. For instance, in Figure 4, communications between vTensors 3, 4 of A and vTensors 5, 6 of B can be materialized using the more efficient alltoall primitive. nnScaler employs simple pattern matching to identify appropriate collective primitives for each pTensor and its associated vTensors.

7 Implementation and Experiences

We implemented nnScaler based on PyTorch [43] with 24K lines of Python code. nnScaler takes a PyTorch model developed for a single device, and converts it into an intermediate graph representation (IR). After the transformation, the spatial-temporal scheduling, and the insertion of communications and tensor manipulation operations specified in a parallelization plan, each device will receive a sub-graph represented by the IR. nnScaler then converts the sub-graph back to a PyTorch code file. And PyTorch runs the code files (i.e., using `torchrun`) in parallel for distributed training.

To support a wide range of PyTorch models, nnScaler implements an augmented graph converter based on TorchFX [55], comprising 2,243 lines of Python code. This converter combines TorchFX's symbolic execution with value tracing of `torch.jit.trace` to handle control flow, which is a typical barrier when converting PyTorch models to TorchFX. By default, PyTorch models usually contain only the forward pass. nnScaler automatically completes the backward pass using autograd functionality with the chain rule [40]. So far, nnScaler has successfully converted 26319 out of 31301 (84.1%) PyTorch models from HuggingFace [25] Natural Language Processing tasks. The conversion failures are mainly due to unsupported operators, e.g., the custom operators designed for specific models. We are actively exploring way to support more operators, along with their corresponding transformation algorithms.

7.1 Experiences

nnScaler has been used by multiple projects across different teams in Microsoft to support the pretraining and fine-tuning of next generation DNN models on several generations of NVIDIA and AMD GPUs. This includes RetNet [51], YOCO [52], LongRoPE [16], Phi-3 series [7][1], and a large science foundation model consisted of a transform-based model combined with a graph neural network. The model size ranges from 3 billion to 92 billion parameters.

The decision to use nnScaler is based on two key factors. First, incorporating new models into existing distributed training frameworks presents intricate engineering challenges. This involves tasks such as the parallelization of the new modules, identifying suitable partition options, and ensuring the end-to-end training correctness, which includes tasks like data loading, gradient normalization (*gnorm*) [12], and optimizer. This process typically takes two experienced engineers about two months to complete. Compounding to the problem, existing parallelization plans often do not works well on new models, resulting in unsatisfactory Model FLOPs Utilization (MFU). Second, the research on new models often requires changes to model architectures, configurations, and training settings. This, in turn, may necessitate further adjustments to be made to the parallelization plan for efficient training, a daunting task for machine learning researchers. nnScaler precisely targets these pain points. Since nnScaler separates codes for the logical model from codes for the parallelization plan, it enables a separation of concerns: model developers can focus on model architecture innovations while system developers can study better parallelization plans. Moreover, our collaboration with these teams have yielded a number of insights, which will be discussed next.

Debugging nnScaler. nnScaler offers great flexibility in model training, but the new primitives and constraints also contribute to increased system complexity, rendering certain parallelization plans error-prone. nnScaler enables a modular approach to debugging system problems, where a new,

[1]nnScaler is used in some post training steps for the long context version of Phi-3, not for the model pretraining.

less-studied sub-graph generated by a new constraint can be replaced with a well-tested constraint. For instance, nnScaler can selectively apply data parallelism, which is less likely to have bugs, to a portion of the model while maintaining the existing parallelization plan for the rest of the model unchanged. This adjustment does not require model code modification; it simply configures the pre-build parallelization plan. By iteratively changing the suspected modules in the plan, it facilitates the identification of the problematic module.

Model accuracy. Achieving high model accuracy is the ultimate goal of model training. However, oftentimes, even a small bug in the training framework or model code can result in a degraded accuracy. Further complicating matters is that while the situation may appear normal in the early training stage, the loss curve tends to deteriorate over an extended training period (e.g., thousands of steps for a 7B LLM) and may eventually diverge. Directly comparing the loss and gradient values with well-tested training plans like data parallelism is impractical. For example, as the reduce operations (e.g., `matmul` or `allreduce`) in more complicated plans introduce drifts in floating-point values due to different orders of summation [20], which is an expected behavior. This makes it difficult to discern if it is an expected numeric deviation or a semantic bug. With respect to this issue, nnScaler firstly evaluates the correctness of a large-scale parallelization plan for a model by reducing the model's hidden dimension to fit the training in a single device. This makes debugging the correctness a much easier task. The model change is easily achievable by slight changes in the model code, thanks to the clean separation between the model code and training code. Subsequently we applied the searched parallelization plan to the reduced model and then assessed the overlap of the loss and gnorm curves with their counterparts in the well-tested data parallelism training. We observed that the gnorm curve is a good indicator, amplifying divergence at earlier stages and signaling potential bugs in the system.

In-place operators. To improve training performance, in-place operators like `Tensor.add_` update tensors in-place. However, partitioning in-place operators could become problematic. For example, if the partitioning of the in-place operator leads to the cloning of a tensor that originally implements in-place updates, the resulting non-inplace sub-operator would not preserve the original effect of the in-place operator's effect. This is due to a violation of the Static Single Assignment (SSA) form [14] when mixing in-place and non-inplace operators. To avoid this problem, nnScaler follows SSA during graph transformation, then replaces some of the non-inplace operators with their original in-place versions in the later optimization phase.

8 Evaluation

The evaluation of nnScaler covers the expressiveness of parallelization primitives and the search efficiency of parallelization plan with constraints. More importantly, we evaluated the performance of the newly searched parallelization plans on real-word models to demonstrate the effectiveness of the entire system in achieving efficient parallelization of new models and settings. In summary, the evaluation results show that:

- The parallelization primitives in nnScaler can construct various parallelization plans, including both existing handcrafted ones (§8.1) and newly innovated ones, as introduced in this paper(§8.2).
- End-to-end evaluation of the three novel parallelization plans on SwinTransformer, T5, and AlphaFold2 shows up to 3.5×, 2.5×, 1.4× speedup, respectively, compared to the baselines of Megatron-LM [39], Alpa [65], DeepSpeed [47], and DAP [13]. (§8.3)
- Parallelization space with constraints helps nnScaler quickly discover efficient plans, resulting in an 11.7× search speedup compared to the searches without constraints.

8.1 Expressiveness of Plan Construction

We evaluated the expressiveness of the three primitives for plan construction by implementing popular handcrafted parallelization plans listed in Table 8. These plans can be decomposed into operator transformation, placement and ordering, which is well aligned with the three primitives in Table 1. 14 out of 17 parallelization plans can be successfully supported by nnScaler. The parallelization plans under SPMD are implemented through `op-trans`. Data and flexible tensor parallelism can be easily supported. Transformer Parallelism and DAP are handcrafted tensor parallelisms for Transformer and AlphaFold2, respectively. Sequence Parallelism and ZeRO stage-3 are special tensor parallelisms, that decouple the partitioning of the operator and its input tensor to optimize memory usage. nnScaler supports them by inserting an `identity` operator between the input tensor and its operator through `op-trans`, facilitating easy decoupling.

The parallelization plans under MPMD are different types of handcrafted pipeline parallelism. They can be supported using `op-order` without implementing a new execution engine. Notably, nnScaler does not support PipeDream due to its asynchronous training method, as nnScaler respects the original training semantics of a model. For TeraPipe, nnScaler currently lacks access to concrete values in tensors, preventing it from determining data dependency at the token level (*i.e.,* tensor masks), a requirement for TeraPipe. In the future, nnScaler can implement TeraPipe through instrumentation tools for deep learning models like [21].

Beyond parallelisms, nnScaler also accommodates memory optimization techniques (e.g., recompute, swap) and the overlapping of computations and communications. Its support of recompute relies on a customized `algo` of `op-trans`

Categories	Mechanisms	Support
SPMD Parallelism	Data Parallelism [1]	✓
	Flexible Tensor Parallel [26, 59, 61]	✓
	Transformer Parallelism [50]	✓
	DAP [13]	✓
	Sequence Parallelism [29]	✓
	ZeRO [47]	✓
MPMD Parallelism	1F1B [18, 50]	✓
	GPipe [24]	✓
	Chimera [30]	✓
	PipeDream (Async) [38]	×
	TeraPipe [31]	×
Memory Optimizations	Gradient Accumulation [60]	✓
	Recompute [11]	✓
	Chain-recompute [28]	✓
	Swap [23]	✓
Overlapping	ByteScheduler [42]	×
	All-reduce Overlap [49]	✓*

Table 8: Supported parallelization plans. '*' requires additional co-scheduling of computation and communication at runtime.

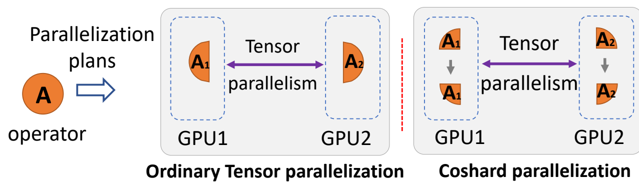


Figure 5: Coshard plan found by nnScaler.

to transform an operator to its recompute version, similar to `torch.utils.checkpoint` in PyTorch [6], while deferring the materialization to the compilation phase. nnScaler does not support ByteScheduler [42], which overlaps two consecutive mini-batches. This is because the boundary of transformation and scheduling in nnScaler is a single mini-batch, though it could potentially be extended to multiple mini-batches.

8.2 Plan Search Results

With the new constraints described in §4.2, nnScaler searches within each constructed space and discovers three novel parallelization plans that show superior training performance.

Coshard. Figure 5 illustrates Coshard, which is used for models with large tensors like SwinTransformer. It can co-exist with tensor parallelism to reduce peak memory of activation tensors. For example, A_1 is partitioned into two, placed on the same device, and executed sequentially. After applying recompute of A_1, the peak activation size of A_1 is halved. Due to the reduced peak memory, tensor parallelism can now span fewer devices (e.g., from 8-way to 4-way), reducing communication cost.

Interlaced pipeline. Figure 6 shows the pipeline schedule searched under the constraints specified in Table 6. The embedding layer is partitioned across four devices using tensor parallelism. The remaining components (i.e., non-embedding layers) are separated to distinct device groups following `staged_spmd`. During the ordering search, all the layers compose into a schedule that resembles executing embedding layers and an 1F1B-like schedule following a time-sharing pattern. There are two columns with 0-th embedding because the embedding layer is used twice, one at the beginning of the model and the other at the end. Thanks to the scheduling search, the pipeline can reach a stable phase with zero bubbles as shown on the right of the figure.

3F1B pipeline. Figure 2 displays the timeline for the 3F1B pipeline which has been described in §4.2. The constraints outlined in Table 7 define how forward and backward passes interleave in the stable state of the pipeline. The schedule for the warm-up and cool-down phases remains unspecified. These phases are tailored through the search process.

8.3 End-to-End Performance

We evaluate the three new parallelization plans on Swin-Transformer, T5, and AlphaFold2, respectively, with different model configurations and on varying number of GPUs.

8.3.1 Experimental Setup

Machine configurations. Our evaluation is performed on DGX-2 clusters with 32 NVIDIA Tesla V100 (32GB) GPUs. Each server is equipped with 16 GPUs that are connected via NVLink [4]. Servers are interconnected with 8 InfiniBand 100 Gbps network adapters. All the servers are installed with NCCL 2.14 [3] and PyTorch v2.0.1 [43]. As 8 × 100 Gbps InfiniBand is a high-end hardware configuration, we also demonstrate the training performance on commodity hardware that is prevalent in many organizations [8]. Specifically, we conducted experiments on DGX-1 clusters with 32 NVIDIA Tesla V100 (32GB) GPUs, each equipped with 1 InfiniBand 100 Gbps network adapter in §8.3.5.

Model configurations. Table 9 summarizes the configurations of SwinTransformer, T5, and AlphaFold2, each of which has four different model configurations ranging from small models to large ones. For each configuration, we list its number of parameters, number of layers, hidden dimensions, and number of heads. For example, <1.8B, 32 layers, hidden size 512, 16 heads> is a configuration for SwinTransformer. The four small to large configurations for each model run on 4, 8, 16, and 32 GPUs respectively.

Baseline systems. We compared nnScaler with three popular distributed training systems: 1) Megatron-LM [39] is designed to train transformer-based models, which hierarchically combines pipeline parallelism with data and tensor parallelism. For pipeline parallelism, it evenly partitions model layers into

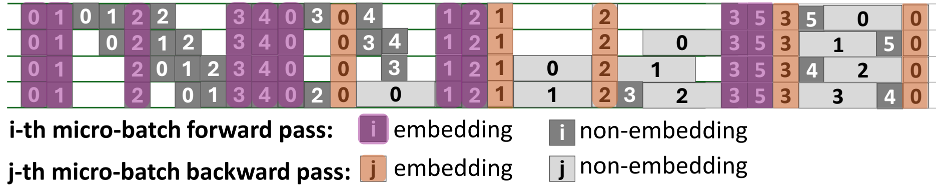


Figure 6: Interlaced pipeline plan found by nnScaler.

Model	SwinTransformer [34]			
Param# (B)	1.8	5.0	10	27
Layer#	32	40	48	56
Hidden	512	768	1K	1.5K
Head#	16	24	32	32
Model	T5 [45, 62]			
Param# (B)	3.9	11	21	47
Layer#	48	64	64	64
Hidden	2K	3K	4K	6K
Head#	32	48	64	96
Model	AlphaFold2 [27]			
Param# (B)	0.087	0.93	2.4	3.2
Layer#	48	64	96	128
Hidden	256	512	1K	1K
Head#	8	16	32	32

Table 9: Model architecture with the increasing number of GPUs. K: thousand. B: billion.

pipeline stages, and each stage can be further applied with data and tensor parallelism. 2) Alpa [65] is an automatic parallelization system for deep learning models under the 3D parallelization space. Its search algorithm and training system are currently based on TensorFlow. To conduct a side-by-side comparison, we implemented the Alpa's search algorithm as a policy in nnScaler. 3) DeepSpeed [47] is a distributed training system similar to Megatron-LM. It supports pipeline, data, and tensor parallelism, Additionally, it incorporates techniques including ZeRO [46] and ZeRO-Offload [48] to optimize GPU memory usage. ZeRO mainly optimizes memory usage of optimizer states by keeping a single copy in data parallelism. ZeRO-Offload offloads weights to CPU memory to reduce the memory pressure of GPU and retrieves them after they are used. It does not support offloading activation tensors.

Neither Megatron-LM nor DeepSpeed features automatic search for parallelization plans in their supported parallelization space. Therefore, we manually found the best-performing plans for them by separately traversing the degrees of pipeline, tensor, and data parallelism respectively. In all the following experiments, we applied layer-wise recompute [11] to reduce the memory consumption of activation tensors. Following the common practice [29, 65], we used the aggregated effective TFLOPS as our performance metric.

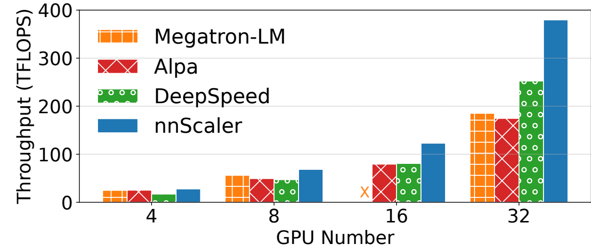


Figure 7: End-to-end training throughput of SwinTransformer. "×" denotes failure due to out of memory.

8.3.2 Results of SwinTransformer

Figure 7 illustrates the end-to-end training throughput of SwinTransformer on four systems. Both Megatron-LM and Alpa use pure tensor parallelism for all model configurations due to the substantial size of activation tensors is huge (e.g. 21GB for the first transformer layer for a 5.0B model), even with recompute applied. DeepSpeed employs ZeRO-Offload and ZeRO stage3 to optimize memory usage. Therefore, DeepSpeed is able to apply 2-way tensor parallelism for the 4 GPUs setting and 4-way tensor parallelism for the remaining three settings. Data parallelism is further applied to scale out across all the available GPUs. nnScaler applies Coshard on the first four layers (Attention+MLP) of SwinTransformer, because these layers occupy a large proportion of memory due to activation tensors. nnScaler applies 2-way, 2-way, 4-way, and 8-way tensor parallelism to the four configurations, respectively, combined with 2-way, 4-way, 4-way, and 4-way pipeline parallelism, respectively. Coshard has 6 partitions sequentially executed on each GPU for the 8 GPUs setting and 4 partitions for the remaining three settings. As shown in Figure 7, nnScaler is 1.2×, 1.5×, and 1.5× faster than DeepSpeed on 8, 16, and 32 GPUs, respectively. Although with ZeRO stage3 to reduce the degree of tensor parallelism to control the communication overheads, ZeRO stage3 still introduces heavy communication costs for weights on the critical path of forward and backward passes, especially when it is applied on 32 GPUs, which involves cross-node communication. In contrast, nnScaler applies Coshard to reduce peak memory, making it possible to use a less degree of tensor parallelism, which reduces communication costs.

Coshard is also used in the long-context post training of the Phi-3 series models to reduce the excessive memory usage due to the long context window [16].

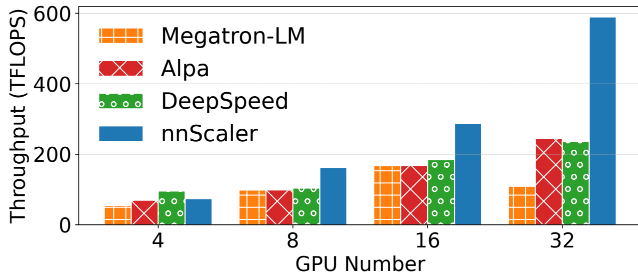


Figure 8: End-to-end training throughput of T5.

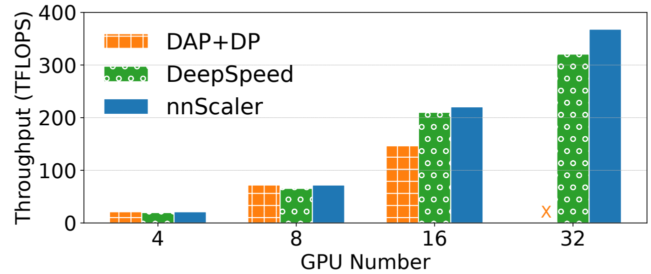


Figure 9: End-to-end training throughput of AlphaFold2. "×" denotes failure due to out of memory.

8.3.3 Results of T5

Figure 8 illustrates the end-to-end training throughput of T5. Megatron-LM uses 2-way tensor parallelism with 2-way pipeline parallelism for 4 GPUs, and uses pure tensor parallelism for 8, 16, 32 GPUs. For 4 GPUs, Alpa uses 3-way pipeline parallelism with the middle stage applying 2-way tensor parallelism. It *must* use pure tensor parallelism for 8, 16, and 32 GPUs due to large memory consumption. As T5 of 3.9B parameters is relatively small, DeepSpeed can use data parallelism with ZeRO stage3 for 4 GPUs. It applies 4-way tensor parallelism for 8, 16, and 32 GPUs, with ZeRO-Offload and ZeRO stage3 applied. Additionally, data parallelism is further applied to scale out to all the available GPUs. nnScaler applies the interlaced pipeline. The large embedding layer uses tensor parallelism on all the available GPUs. The remaining layers apply 4-way pipeline parallelism, with each stage applied 1-way, 2-way, 4-way, and 8-way tensor parallelism for 4, 8, 16, and 32 GPUs, respectively.

nnScaler performs 1.5×, 1.6×, and 2.5× better than DeepSpeed for 8, 16, and 32 GPUs respectively. Megatron-LM and Alpa have a low performance because the high degrees (e.g., 32) of tensor parallelism introduces high communication overheads, especially when the tensor parallelism spans more than one node. This is why Megatron-LM performs much worse with 32 GPUs. As Alpa searches for suitable partition options for tensor parallelism, many operators (e.g., dropout or layernorm) are replicated across nodes to reduce communication costs, and so it performs better than Megatron-LM. Although DeepSpeed has a lower degree of tensor parallelism, its performance is only comparable to Alpa because DeepSpeed uses ZeRO-Offload and ZeRO stage3 to make lower degrees of tensor parallelism feasible. ZeRO-Offload introduces high overheads due to the offloading of large embedding weight (e.g., 12GB in the 21B model). ZeRO stage3 also introduces high communication costs, such as the online gathering of (embedding) weights on the critical path. This shows the effectiveness of the proposed interlaced pipeline on models like T5, compared to conventional approaches like tensor parallelism, ZeRO-Offload, and ZeRO stage3. Note that to highlight the advantage of interlaced pipeline, nnScaler tentatively disables ZeRO in the experiments in Figure 8. And nnScaler still outperforms others in most cases except for the 4-GPU case when T5 is small enough to fit-in memory after applying data parallelism with ZeRO (i.e., DeepSpeed's plan).

8.3.4 Results of AlphaFold2

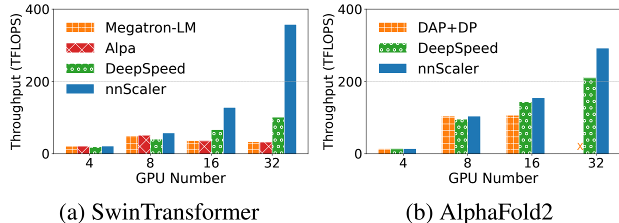
Figure 9 shows the end-to-end training throughput of AlphaFold2. In this experiment, we compared nnScaler with two baselines. One is DAP [13], which is a handcrafted tensor parallelism specifically designed for AlphaFold2. We also applied data parallelism to scale out DAP, referred to as DAP+DP. For 4 and 8 GPUs, DAP+DP uses pure data parallelism since the models are small. It uses 4-way tensor parallelism with 4-way data parallelism for 16 GPUs. The other baseline is DeepSpeed. As the model sizes are much smaller than those of SwinTransformer and T5, the application of ZeRO-Offload is not necessary. DeepSpeed uses pure data parallelism for 4, 8, and 16 GPUs with ZeRO stage3, and uses 2-way tensor parallelism with 16-way data parallelism for 32 GPUs. nnScaler also uses pure data parallelism for 4 and 8 GPUs. It applies the 3F1B pipeline for 16 and 32 GPUs. For 16 GPUs, nnScaler uses 4-way pipeline parallelism with 4-way data parallelism, while for 32 GPUs it uses 2-way tensor parallelism with 2-way pipeline parallelism and 8-way data parallelism.

nnScaler performs 1.5× better than DAP+DP on 16 GPUs and 1.1× better than DeepSpeed on 32 GPUs. DeepSpeed performs better than DAP+DP on 16 GPUs, because the activation tensors in AlphaFold2 are large, and the communication of activation tensors using 2-way tensor parallelism is more efficient than that using 4-way tensor parallelism. nnScaler performs better than DeepSpeed because the customized 3F1B pipeline reduces communication costs. The training conducted on multiple nodes, which is common for large model training, amplifies the advantage of pipeline parallelism.

8.3.5 Experiments on Less Powerful Hardware

To demonstrate the effectiveness of the new parallelization plans and understand how different hardware affects training performance, we evaluate SwinTransformer and AlphaFold2

in the DGX-1 cluster. As shown in Figure 10a, nnScaler is 1.9× and 3.5× faster than DeepSpeed on 16 and 32 GPUs, respectively. Compared with data shown in Figure 7, for 32 GPUs, the performance of nnScaler is degraded by 6%, while that of DeepSpeed, Alpa, and Megatron-LM is degraded by 60%, 82% and 82%, respectively. The degradation of nnScaler is smaller because the parallelization plan (i.e., Coshard) used by nnScaler optimizes the communication cost, and thereby it tolerates the changes in communication bandwidth. Figure 10b shows the results of AlphaFold2 on DGX-1. The relative performance gain of nnScaler is also improved to 1.1× and 1.4× over DeepSpeed on 16 and 32 GPUs, respectively. The lower bandwidth cross nodes in DGX-1 further amplifies the advantage of pipeline parallelism, rendering the 3F1B pipeline much faster than tensor parallelism in DAP+DP and DeepSpeed. These experiments indicate that with the flexible customization of parallelization plans and automatic plan search, nnScaler can adapt more flexibly to changes in hardware.



(a) SwinTransformer (b) AlphaFold2

Figure 10: End-to-end training throughput on DGX-1. "×" denotes the failure of training due to out-of-memory.

8.4 Search Efficiency with Constraints

Algorithm 1 suggests that the parallelization plan search cost in nnScaler consists of: (1) operator transformation and placement cost (i.e., line 1-4 in Algorithm 1), and (2) operator temporal ordering cost (i.e., line 5 in Algorithm 1). Figure 11 illustrates the end-to-end search cost, as well as the breakdown time of the three customized spaces defined in §4.2 for different model configurations using the policy illustrated in §5. The search on SwinTransformer's space takes less than 150s. The search time increases with the increase of model size as the number of operators increases. The ordering search for T5 takes around 150s due to an absence of constraints on the ordering in T5's space. There is almost no search cost of the ordering in SwinTransformer and AlphaFold2. For SwinTransformer, the order is largely determined by data dependencies, and for AlphaFold2, the ordering constraints greatly reduces the space.

Figure 12 further shows the temporal ordering search time of the 3F1B schedule with and without constraints. The left figure shows that the search time increases exponentially with the increase of stage number. However, with constraints ap-

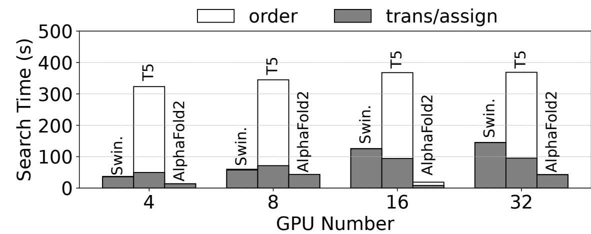


Figure 11: End-to-end search cost of each model.

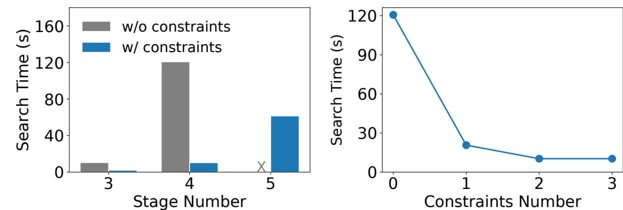


Figure 12: Search time of the 3F1B schedule with and without constraints. "×" denotes a search time exceeding one hour.

plied, the search time is kept within 60s, resulting in 11.7× speedup in finding the efficient temporal ordering for 4 stages. This is attributed to the temporal ordering constraints in Table 7, where the ordering constraints of independent forward and backward operators from different micro-batches are explicitly specified, leading to a significantly reduced search space exposed to the search algorithm (i.e., Tessel). For the case of 4 stages, the right figure further shows the search time as each ordering constraints from Table 7 is applied one by one. The first constraint reduces the search time by 100s. The second constraint further reduces it by 50% of search time. This demonstrates the importance of constraints.

9 Related Work

Existing parallelization search spaces. Recently, data, tensor, and pipeline parallelisms [1, 18, 24, 30, 46] have been widely used in distributed DNN training. Various memory optimizations [11,23,28] have also been adopted to exploit large-scale model training under GPU memory constraints. Systems such as Megatron-LM [29,39,50], DeepSpeed [47], Piper [54], Unity [57], and Alpa [65], combine multiple parallelisms and memory optimizations to accelerate distributed DNN training. However, these solutions fall short in because they rely on empirical parallelism configurations and have limited execution scheduling choices. Thus, despite their successful applications on existing training workloads, they still fail to fully utilize hardware capabilities. In contrast, nnScaler provides a different approach to parallelization, supporting the expression of parallelization sub-spaces with fine-grained transformation and scheduling primitives. Consequently, nnScaler is compatible with them as all these solutions can be achieved

using particular constraints. In additional, nnScaler is able to support more flexible and efficient parallelization plans that extend beyond the aforementioned parallelization sub-spaces, which is considerably crucial for continuously evolving DNN models.

Explorations on specific parallelization plans. Parallelization strategies tailored for specific scenarios play a crucial role in optimizing the performance of parallel computing frameworks. For instance, Transformer Parallelism [50], DAP [13], and Sequence Parallelism [29] are designed for specific model architectures, showcasing a nuanced approach to parallelization. To address the need for optimized pipeline orchestration, innovative scheduling strategies have been proposed by GPipe [24], 1F1B [18, 50], and Chimera [30]. Furthermore, optimizations such as Gradient Accumulation [60], Recompute [11], Chain-recompute [28], Swap [23], and All-reduce Overlap [49] specifically target improvements in memory or communication efficiency. These strategies can be seamlessly incorporated into nnScaler's plan with appropriate constraints, eliminating the need for a comprehensive system overhaul and demonstrating the platform's adaptability.

Parallelization plan search and others. To improve training performance with combined parallelisms, DNN systems [22, 26, 37, 54, 59, 63, 65] use different searching techniques to find efficient parallelism configurations. Most recently, Alpa [65] leverages both integer programming and dynamic programming solvers, and Tessel [32] enables the exploration of schedule search in pipeline parallelism, significantly harnessing performance potential beyond manually crafted pipeline schedules. nnScaler, as a parallelization plan engine that emphasizes customizing the parallelization space through constraints, is complementary to the above algorithms and can leverage them to speedup the search within a customized space.

Kernel fusion and tuning optimizations. Besides efficient parallelization plans, kernel fusion and tuning [10, 15, 36, 66] can also improve execution efficiency on a device by fusing multiple consecutive operators into a single more performant GPU kernel. For instance, Flash-Attention [15] fuses multiple operations within the attention layer into a single kernel to improve performance with reduced I/O. These techniques are complementary to nnScaler as they can be applied after nnScaler partitions computation across devices, to further enhance the local computation efficiency on each device.

10 Conclusions

nnScaler is a framework that enables domain experts to leverages three primitives, `op-trans`, `op-assign`, and `op-order`, along with constraints to construct arbitrary search spaces for parallelization plans given any DNN model. This approach represents a more general abstraction to describe both existing parallelization search spaces and new spaces. Experiments show that nnScaler is able to construct new spaces that lead to the discovery of new parallelization plans for deep learning training on emerging DNN models as well as mainstream models, significantly outperforming existing plans.

11 Acknowledgements

We sincerely thank our shepherd and all the anonymous reviewers for their valuable feedback. We also thank Madan Musuvathi, Guodong Liu, Xiaoxiang Shi and Jun Huang for their suggestions and help.

References

[1] Distributed Data Parallelism. https://pytorch.org/docs/stable/notes/ddp.html. [Online; accessed Sep. 2022].

[2] Introducing ChatGPT. https://openai.com/blog/chatgpt. [Online; accessed Nov. 2023].

[3] NVIDIA collective communications library. https://developer.nvidia.com/nccl. [Online; accessed Aug. 2021].

[4] NVIDIA NVLINK. http://www.nvidia.com/object/nvlink.html.

[5] Project nnScaler. https://github.com/microsoft/nnscaler.

[6] PyTorch Team. https://pytorch.org/docs/stable/checkpoint.html. [Online; accessed Aprl. 2022].

[7] Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024.

[8] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, et al. The falcon series of open language models. *arXiv preprint arXiv:2311.16867*, 2023.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[10] 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)*, pages 578–594, 2018.

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

[12] Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee, and Andrew Rabinovich. Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. In *International Conference on Machine Learning (ICML)*, pages 794–803. PMLR, 2018.

[13] Shenggan Cheng, Ruidong Wu, Zhongming Yu, Binrui Li, Xiwen Zhang, Jian Peng, and Yang You. Fastfold: Reducing alphafold training time from 11 days to 67 hours. *arXiv preprint arXiv:2203.00854*, 2022.

[14] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[15] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:16344–16359, 2022.

[16] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. Longrope: Extending llm context window beyond 2 million tokens. *arXiv preprint arXiv:2402.13753*, 2024.

[17] Richard Evans, Michael O'Neill, Alexander Pritzel, Natasha Antropova, Andrew Senior, Tim Green, Augustin Žídek, Russ Bates, Sam Blackwell, Jason Yim, Olaf Ronneberger, Sebastian Bodenstein, Michal Zielinski, Alex Bridgland, Anna Potapenko, Andrew Cowie, Kathryn Tunyasuvunakool, Rishub Jain, Ellen Clancy, Pushmeet Kohli, John Jumper, and Demis Hassabis. Protein complex prediction with alphafold-multimer. *bioRxiv*, 2021.

[18] 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 (PPoPP)*, pages 431–445, 2021.

[19] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 551–568, 2021.

[20] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)*, 23(1):5–48, 1991.

[21] Yue Guan, Yuxian Qiu, Jingwen Leng, Fan Yang, Shuo Yu, Yunxin Liu, Yu Feng, Yuhao Zhu, Lidong Zhou, Yun Liang, Chen Zhang, Chao Li, and Minyi Guo. Amanda: Unified instrumentation framework for deep neural networks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[22] Ubaid Ullah Hafeez, Xiao Sun, Anshul Gandhi, and Zhenhua Liu. Towards optimal placement and scheduling of dnn operations with pesto. In *Proceedings of the 22nd International Middleware Conference*, pages 39–51, 2021.

[23] 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 (ASPLOS)*, pages 1341–1355, 2020.

[24] 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. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 103–112, 2019.

[25] Hugging Face Community. Model hub – natural language processing. https://huggingface.co/models. [Online; accessed Jun. 2023].

[26] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *SysML*, 2019.

[27] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

[28] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. In *9th International Conference on Learning Representations, (ICLR), Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

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

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

[31] 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. In *International Conference on Machine Learning (ICML)*, pages 6543–6552. PMLR, 2021.

[32] Zhiqi Lin, Youshan Miao, Guanbin Xu, Cheng Li, Olli Saarikivi, Saeed Maleki, and Fan Yang. Tessel: Boosting distributed execution of large dnn models via flexible schedule search. *arXiv preprint arXiv:2311.15269*, 2023.

[33] Guodong Liu, Youshan Miao, Zhiqi Lin, Xiaoxiang Shi, Saeed Maleki, Fan Yang, Yungang Bao, and Sa Wang. Aceso: Efficient parallel dnn training through iterative bottleneck alleviation. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*, pages 163–181, 2024.

[34] Ze Liu, Han Hu, Yutong Lin, Zhuliang Yao, Zhenda Xie, Yixuan Wei, Jia Ning, Yue Cao, Zheng Zhang, Li Dong, et al. Swin transformer v2: Scaling up capacity and resolution. *arXiv preprint arXiv:2111.09883*, 2021.

[35] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. *arXiv preprint arXiv:2103.14030*, 2021.

[36] 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)*, pages 881–897, 2020.

[37] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. Paddlepaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing*, 1(1):105–115, 2019.

[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 (SOSP)*, pages 1–15, 2019.

[39] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–15, 2021.

[40] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

[41] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 891–905, 2020.

[42] 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 (SOSP)*, pages 16–29, 2019.

[43] PyTorch Team. PyTorch. https://pytorch.org//. [Online; accessed Mar. 2022].

[44] PyTorch Team. PyTorch Distributed Communication Package. https://pytorch.org/docs/stable/distributed.html. [Online; accessed Nov. 2023].

[45] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 2020.

[46] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training a trillion parameter models. *arXiv preprint arXiv:1910.02054*, 2019.

[47] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 3505–3506, 2020.

[48] 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. In *2021 USENIX Annual Technical Conference (USENIX ATC)*, pages 551–564, 2021.

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

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

[51] Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models, 2023.

[52] Yutao Sun, Li Dong, Yi Zhu, Shaohan Huang, Wenhui Wang, Shuming Ma, Quanlu Zhang, Jianyong Wang, and Furu Wei. You only cache once: Decoder-decoder architectures for language models. *arXiv preprint arXiv:2405.05254*, 2024.

[53] Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307*, 2020.

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

[55] PyTorch Team. TorchFX. `https://pytorch.org/docs/stable/fx.html`.

[56] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

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

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

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

[60] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC)*, pages 84–97, 2016.

[61] 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.

[62] Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua, and Colin Raffel. mt5: A massively multilingual pre-trained text-to-text transformer. *arXiv preprint arXiv:2010.11934*, 2020.

[63] 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 (NeurIPS)*, 33:906–917, 2020.

[64] Bo Zheng, Li Dong, Shaohan Huang, Saksham Singhal, Wanxiang Che, Ting Liu, Xia Song, and Furu Wei. Allocating large vocabulary capacity for cross-lingual language model pre-training. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 3203––3215, 2021.

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

[66] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. Roller: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 233–248, 2022.