TAPAS: Fast and Automatic Derivation of Tensor Parallel Strategies for Large Neural Networks

Ziji Shi*
National University of Singapore
Alibaba Group
Singapore
ziji.shi@u.nus.edu

Le Jiang
Alibaba Group
Hangzhou, China
jiangle.jl@alibaba-inc.com

Ang Wang
Alibaba Group
Hangzhou, China
wangang.wa@alibaba-inc.com

Jie Zhang
Alibaba Group
Hangzhou, China
wanglin.zj@alibaba-inc.com

Chencan Wu
Alibaba Group
Hangzhou, China
danguge@buaa.edu.cn

Yong Li
Alibaba Group
Hangzhou, China
jiufeng.ly@alibaba-inc.com

Xiaokui Xiao
National University of Singapore
Singapore
xkxiao@nus.edu.sg

Wei Lin
Alibaba Group
Hangzhou, China
weilin.lw@alibaba-inc.com

Jialin Li
National University of Singapore
Singapore
lijl@comp.nus.edu.sg

Abstract

Tensor parallelism is an essential technique for distributed training of large neural networks. However, automatically determining an optimal tensor parallel strategy is challenging due to the gigantic search space, which grows exponentially with model size and tensor dimension. This prohibits the adoption of auto-parallel systems on larger models.

We observe that neural networks usually contain repeated substructures, and build an automatic parallelism framework named TAPAS that eliminates redundant search efforts. TAPAS employs a divide-and-conquer approach that efficiently folds the search space by identifying those unique substructures. As a result, it runs at sub-linear complexity concerning the model size, making it a scalable solution for training large-scale networks. Our evaluations demonstrate that TAPAS outperforms the state-of-the-art automatic parallelism frameworks by up to 160× in search speed on a wide range of models, and the performance of derived strategies is competitive or even better compared with the expert-engineered Megatron-LM library.

CCS Concepts

• **Computing methodologies** → **Artificial intelligence**; • **Computer systems organization** → *Distributed architectures.*

Keywords

Automatic Parallelism, Distributed Training

ACM Reference Format:
Ziji Shi, Le Jiang, Ang Wang, Jie Zhang, Chencan Wu, Yong Li, Xiaokui Xiao, Wei Lin, and Jialin Li. 2025. TAPAS: Fast and Automatic Derivation of Tensor Parallel Strategies for Large Neural Networks. In *Proceedings of 54th International Conference on Parallel Processing (ICPP '25).* ACM, New York, NY, USA, 12 pages. https://doi.org/3754598.3754677

*Work done during internship at Alibaba Cloud.

ICPP '25, San Diego, CA

ACM ISBN 979-8-4007-2074-1
https://doi.org/3754598.3754677

1 Introduction

Model scaling have been the cornerstones in neural network advancements in recent years, resulting in many powerful and gigantic models. Researchers have observed it that model can perform better by increasing the number of parameter, training on larger datasets, and supplying more compute [17]. This has led to the advancements of many powerful models like DeepSeek-V3 [23], Llama [40], and GPT [3]. However, the advancement of memory capacity in AI accelerators has not kept in pace. Over the last decade, the memory capacity of Nvidia GPUs has only increased by 16 times (from K20 to H100) to reach 80GB, whereas the size of neural networks has expanded by 30,000 times (from AlexNet [19] to GPT-4o). To address this problem, researchers propose model parallelism, where model weights and optimizer stats are sharded across multiple machines during distributed training.

However, several challenges on designing training strategies surface as model size scales up. Firstly, manually specifying the optimal parallel strategy is becoming increasingly difficult. While large-language models (LLMs) are popular, there exists many non-transformer-based models for targeted applications. For instance, U-Net[35] is a "U"-shaped convolutional neural network (CNN) used for image segmentation tasks, particularly in medical imaging. Recommendation system usually adopts a two-tower model architecture[7] for mapping user and item features respectively, where each tower has a different design. Large-scale classification models consist of a feature extraction module and a classification

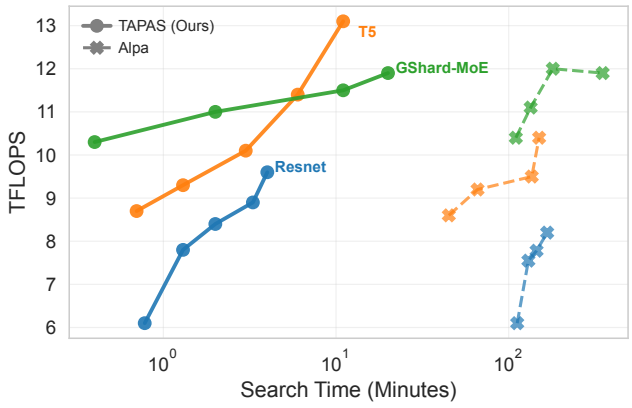

Figure 1: Search time budget vs. training throughput.

module that scales with the number of targets. It is challenging to design a simple strategy that fits all. On top of it, the optimal parallel strategy often requires a combination of multiple parallelization techniques and in-depth knowledge about the underlying system. Therefore, relying solely on expert knowledge to design such strategies is becoming intractable and error-prone.

Furthermore, the number of possible strategies in tensor parallelism grows exponentially with model size. The total number of possible tensor parallel strategies is determined by the Cartesian product of number of tensors and their orders across all tensors in the model. Since recent models can contain thousands of tensors, the total number of possible strategies quickly grows beyond what is feasible to explore exhaustively. For instance, DeepSeek-V3 has 91997 weight tensors, and Llama3.1-405B has 1140 weight tensors. Assuming each tensor has at least three dimensions (batch, sequence, hidden), there are 778.6 trillion and 1.5 billions possible tensor parallel strategies respectively. Therefore, performing exhaustive search over the entire search space can be prohibitively expensive in practice, limiting the adoption of existing automatic model parallel systems.

On top of this, strategy validation becomes computationally expensive as the number of candidate strategies grows. To ensure the new parallel strategy is mathematically equivalent as the original model, each strategy needs to be validated. As the number of candidate strategies scales up, the current dry-run or randomized-testing-based validation method can be a performance bottleneck, and we must design better validation scheme to early-stop on infeasible strategies.

Due to the challenges above, existing state-of-the-art solutions for training auto-parallelism usually suffer from prohibitively long search time. Concretely, Alpa takes 5.8 hours to search on a GShardMoE-2.4B model, yet it is projected to take more than 6 million hours to derive the strategy for a 240 billion counterpart. This scalability limitation makes it challenging to employ auto-parallel systems on large neural network models.

In this work, we propose a novel automatic tensor parallel system that significantly reduces the strategy search time without compromising strategy quality, as shown in Figure 1. It is based on the key observation that repeated substructures (i.e., reused layers/operator groups) are commonplace in network architectures. These repeated substructures can be exploited to effectively *fold* the search space to increase strategy derivation speed. We further show that a parallel strategy, when applied to two similar layers in different parts of the model, exhibits similar resource requirements. This is because the same layers share equivalent communication, computation, and memory access patterns. A direct corollary is that a single parallel schedule, once derived, can be reused for all repeating layers without loss of efficiency.

Guided by this observation, our approach folds the search space by identifying the set of *unique sub-computational graphs*, each representing a unique network. Thereafter, we restrict the search space from the entire computational graph to the set of unique subgraphs, resulting in an exponential decrease in search complexity. To further accelerate the strategy validation process, we adopt an early-stopping framework on the candidate strategy. After that, the remaining challenge is to ensure the optimized subgraphs will combine to form a valid solution. We employ static analysis to verify that the derived sub-strategies are valid and compatible, and that the final parallel strategies maintain mathematical equivalence with the original model. In the end, our system selects the best strategy using a communication-based cost model and reconstructs the parallelized computational graph.

We present TAPAS (Tensor Auto Parallelisation), an automatic parallel framework that efficiently derives tensor parallel strategy for a given neural network. TAPAS requires no expert annotations, and achieves $20-160\times$ search time speedup over the state-of-the-art auto-parallel framework Alpa [51]. With the growing size of foundation models, TAPAS proves to be a scalable option. We further demonstrate that TAPAS can identify a tensor parallel strategy with comparable performance as expert-designed solutions like Megatron-LM [38] or DeepSpeed[33].

We summarize our contribution as the following:

- We identify the problem that automatic searching for a parallel strategy can be slow and inefficient, formulate the prior works under one view, and provide a grounded analysis of the complexity.
- Observing that parallel strategies are similar for similar layers/blocks, we propose a computational graph pruning method that efficiently folds the search space into a limited set of subgraphs.
- We design and implement TAPAS, an automatic tensor parallel framework. TAPAS can discover tensor parallel strategies with better or matching performance than the State-of-the-Art strategies, while being two orders of magnitude faster in searching.

2 Related Work

In this section, we outline the current research landscape for model parallelism and automatic parallelism.

2.1 Model Parallelism

Tensor parallelism and pipeline parallelism are two commonly used approaches for model parallel training.

2.1.1 Pipeline Parallelism (PP). Pipeline parallelism divides a model by layers and assigns each group of layers to a separate device [11,

Framework	Search Space	Search Algorithm	Strategy Validation	Overall
FlexFlow	$N(4E, 4V)$	$O(B)$	$O(V+E)$	$O(BV+BE)$
Alpa	$N(kE, kV)$	Inter-Op: $O(V^2L)$ Intra-Op: $O(E(V+E))$	$O(V+E)$	$O(V^2L(V+E^2))$
TAPAS	$N(\frac{E}{2CL}, \frac{V}{2CL})$	$O(\frac{E+V}{L})$	$O(\frac{E}{L})$	$O(\frac{E+V}{L})$

Table 1: Complexities of selected auto parallel frameworks. E and V are the number of connections and number of vertices in the computational graph respectively, L is the number of layers, C is the frequency of repeated substructure.

14, 26, 29]. A training batch is split into micro-batches so that different devices can compute in an overlapped fashion. The efficiency of PP is determined by the bubble time, where devices are idle due to unsatisfied dependency. To keep bubble time low, each pipeline stage should use similar amounts of memory and computation so that workloads remain balanced[29]. In practice, the heterogeneity in models or layer interdependency may limit even pipeline splitting. For example, large classification models often end with a heavy fully-connected output layer, and encoder–decoder models can suffer from cross-attention dependencies between encoder and decoder stages.

2.1.2 Tensor Parallelism (TP). Tensor parallelism, also called tensor sharding, partitions each layer at the tensor level and places each shard on a different device [27, 36, 38]. When a full weight or activation tensor is required, TP uses collective operations such as all-gather or all-reduce to reassemble the full tensor from its shards.

Compared to PP, TP typically incurs higher communication overhead because all devices in the group must exchange data. In addition, the space of possible TP configurations is much larger than for PP, making it more challenging to find an optimal partitioning.

2.2 Automatic Parallelism

With rich options for parallel strategy, it has become difficult to determine which one to use for distributed training. Some existing works focus on building customized training systems for specific models like embedding model [24, 49], Generative Adversarial Networks [37], Mixture-of-Expert models [12, 28], and Graph Neural Networks [44, 50]. Those systems base their optimizations on model-specific characteristics and thus cannot generalize to new models. Automatic parallelism is a recent line of research on automatically selecting parallel strategies for distributed training with minimal user intervention.

Because of the vast search space of parallel strategies, existing works on automatic parallelism either rely on user annotation or brute-force searches over the all possible candidates.

2.2.1 Directive-based approaches. Directive-based automatic parallelism relies on expert annotations to derive parallel strategies. Those annotations are usually bound to specific model dimensions. For example, Mesh TensorFlow [36] infers the operator partitioning scheme based on user-defined directives to scale single-device programs. Whale [15] allows for incorporating user annotation to perform semi-auto parallelisation for large models and introduces a hardware-aware load balance algorithm. However, directive-based automatic parallelism approaches require users to have a deep understanding of both the system and the model, and the hard-coded user annotations may not be transferable when either the model or system changes.

2.2.2 Search-based approaches. Recent work has proposed fully automatic approaches based on search algorithms to optimize distributed DNN training. For example, Tofu [43] uses a recursive search algorithm to derive a communication-efficient schedule for the CNN and RNN models, but it does not generalize to Transformer-based architectures with many dense MatMul operators. Leveraging the discrete nature of candidate spaces, Flexflow [16] uses Markov-Chain Monte Carlo search to find the best parallel strategy for DNN models. While this approach works well for small-scale neural networks, it cannot scale well on large neural networks without properly engineered heuristics. Alpa [51] adopts a two-level optimization approach: it uses an inter-operator optimization pass to cluster operators, and a secondary intra-operator optimization pass to find tensor parallel strategies within the cluster. Unity [41] represents both parallelisation and algebraic transformations in a unified manner, and uses a hierarchical search algorithm to identify an optimized sequence of graph substitutions.

2.2.3 Challenge. While search-based approaches have demonstrated promising results, they encounter a major obstacle: the exponential growth of the search space leads to a prohibitively long search time. Specifically, for neural networks, each N-dimensional tensor offers N+1 possible strategies: not sharding, or sharding along the N-th dimension. Consequently, for a neural network represented as $G(E, V)$ with V tensors, the number of possible tensor parallel strategies can reach up to $(N+1)^V$. As a result, identifying an optimal sharding strategy is beyond the capabilities of polynomial time algorithms, underscoring the critical challenge of managing the vast search space efficiently.

This exponential increase in strategy space has led to impractical search time to derive parallel schedules for large models, which we will later show in subsection 5.2. The search speed problem has become an emerging bottleneck in training foundation models.

3 Approach

Can we accelerate the derivation by leveraging the insights from the model architecture? In this section, we begin by introducing two common patterns of model scaling (by width and by depth), and the challenges associated with the existing approaches in handling these cases. We then formulate the problem of finding the parallel strategy. In the end, we propose to use graph pruning to reduce the search space.

3.1 Motivating Examples

We review common model scaling techniques, and propose that these techniques can be grouped into two major categories: **scaling on the width** by increasing the dimension of layers (e.g., adding the number of classes, adding attention heads, or increasing the convolutional channels), or **scaling on the depth** by increasing the number of layers.

We give two concrete examples of model scaling below.

In the context of e-commerce, a wide range of merchandise exists, potentially numbering in millions to billions. Consequently, a product image classification model such as ResNet [13] requires an exceptionally wide fully connected (FC) layer to classify it. When the number of classes reaches 100,000, the FC layer will comprise 205M floating point numbers, significantly outweighing the feature extraction module, which stands at a modest 24M parameters.

In the scaling-on-depth scenario, we examine models based on the transformer architecture. Currently, most large language models employ the transformer layer [42], which includes an attention module followed by a feedforward network. In the quest for scale, dense transformer models typically stack more transformer layers on top of each other [2, 9, 10, 48], driven by the observation that larger models usually perform better [21, 39, 46]. Because of the uniformity in model architecture, there exists potential for reusing the sharding pattern discovered for one layer across all transformer layers [27].

In summary, it is increasing difficult to build tailored training systems in response to exponentially growing model sizes. How to *automatically* and *quickly* devise a parallel strategy to train these models? We formulate the problem of the automatic derivation of parallel strategy as a graph transformation problem, and present the challenges in the next section.

3.2 Problem Formulation

We formulate the problem using the graph representation of neural networks. All neural networks can be represented as a directed acyclic graph $G(E, V)$ comprised of L layers. The set of vertices V represents the operators, and the set of edges E represents the data flow from producer to consumer operators. During the forward pass, an edge represents an activation tensor, while in the backward phase, an edge represents a gradient or error tensor. A layer $L_i \in L$ could consist one or more nodes. The training cluster is modeled as $S(m, n)$ where m is the number of worker nodes, and n is the number of accelerators per worker node. A parallel strategy P_G is a graph transformation on G that preserves the mathematical equivalence. The goal is to find an optimal parallel strategy ${P_G}^*$ such that: given any input, the output is equivalent and the training throughput is maximized.

The end-to-end duration to produce an optimal schedule is a critical metric for an auto-parallel system. We identify three main factors that contribute to this overall completion time: the size of the search space, the time complexity of the searching algorithm, and the speed of the evaluation method. Out of them, the search space is the most important factor as it determines the input size of the other two factors.

Observation #1: repeated subgraphs commonly exist in large models. As we see earlier, a major challenge faced by auto-parallel systems is the search space explosion problem. Our key observation is that both scaling by width or depth techniques start with a *base subgraph*, i.e., a group of layers or operators, and expand from it. For instance, large-scale pre-trained language models such as BERT [9] and T5 [30] consist of tens of transformer layers, and multi-class object classification networks like ResNet-50 [13] are made of repeated convolutional layers. In DeepSeek-v3/R1 model, the basic components are MLP, self-attention, and MoE layers. This indicates that we can save much effort by restricting the search on the subgraphs instead of the whole graph, where a lot of search efforts are wasted on identical structures.

Observation #2: the performance of repeated subgraphs is identical across the model. Furthermore, by analyzing expert-engineered parallel schedules [27, 31, 34], we observe that parallel schedules are primarily identical for the same type of layers. The underlying reason is the same layers share the same amount of communication, computation, and memory access patterns. Therefore, their performance should also be similar.

This has motivated us to explore the possibilities of reusing the parallel schedules discovered for the same layer to save search effort. Specifically, we first identify the unique subgraphs through pattern matching, derive the parallel strategy for each subgraph, then apply the same strategy to the rest of the subgraphs.

Another challenge we face is the complexity of parallel implementations of operators. Given the huge variety of operators, with each of them having multiple possible sharding implementations, how to flexibly represent all possible combinations, and ensure the final implementation is correct? To answer this, we adopt an *enumerate-then-validate* approach by representing all possible strategy combinations using a decision tree-like structure, then validating the tree and performing early stopping if necessary. By doing so, we avoid running the strategies and saved efforts on incorrect strategies.

4 Design and Implementation

4.1 Overview

Based on the insights, we design TAPAS, a tensor auto-parallel framework focusing on the strategy derivation speed. `TAPAS` significantly reduces redundant search efforts by shrinking the search space at different levels without compromising the strategy quality.

As depicted in Figure 2, given any neural network, `TAPAS` first converts the graph into groups of operators named GraphNodes (Step ①). `TAPAS` then performs subgraph mining, restricting the search space from the whole graph to the set of unique subgraphs (Step ②). After finding the unique subgraphs, `TAPAS` enters the Strategy Exploration phase by enumerating all possible parallel strategies for each unique subgraph based on the sharding patterns (Step ③). After that, it validates each strategy to ensure the new computational graph can be correctly reconstructed later(Step ④). At the end of the Strategy Exploration phase, all remaining candidate strategies are evaluated using the cost model (Step ⑤). In the end, `TAPAS` takes the best parallel strategy and reconstructs it back to the computational graph for execution on DL framework backends like TensorFlow.

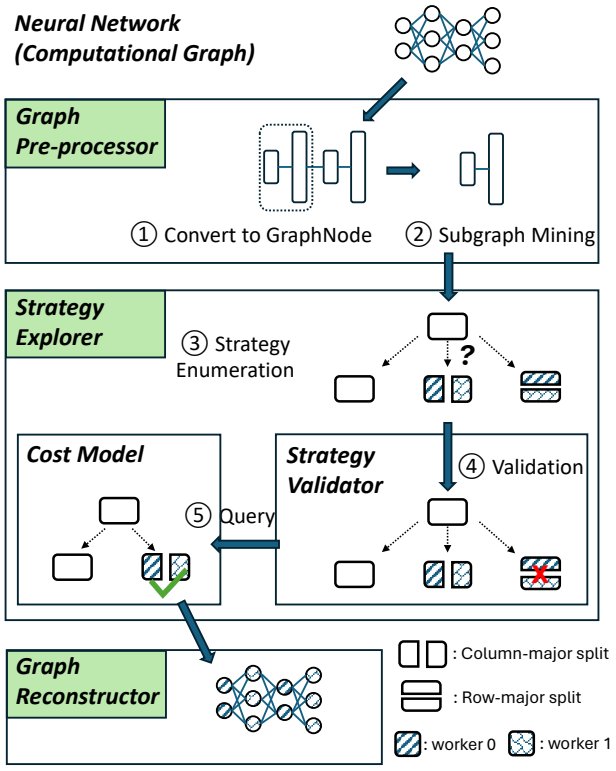

Figure 2: TAPAS system architecture.

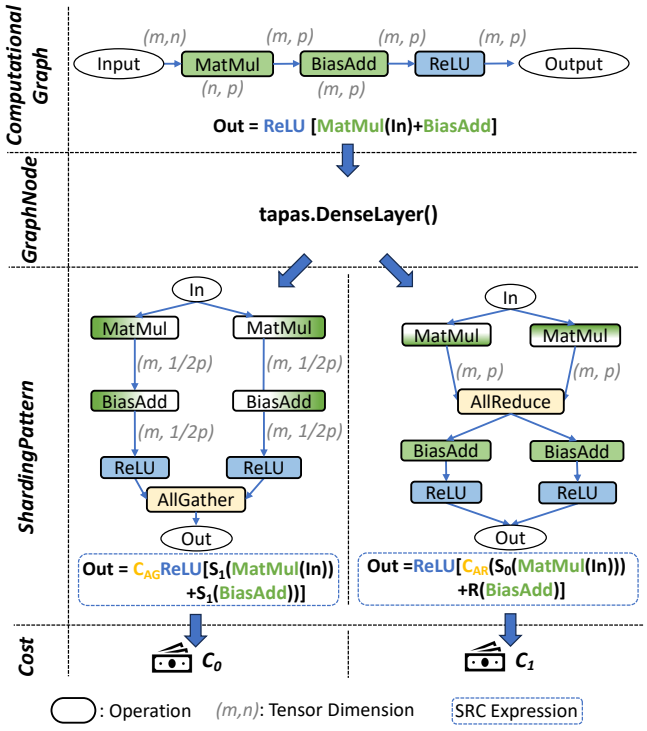

Figure 3: Overview of graph transformation for dense layer.

4.2 Intermediate Representation

TAPAS preprocesses the computational graph into Intermediate Representations (IRs) to facilitate the derivation of parallel strategies. Compared to other deep learning IRs like MLIR HLO [20], TAPAS IR groups operators that are collectively used together. Each group of operators(*GraphNodes*) is associated with a set of possible sharding implementations (*ShardingPatterns*) expressed using the Split-Replica-Communication (SRC) expression. Once the sharding rules are determined, the associated costs are also decided, which are then used to evaluate the strategies.

GraphNode. *GraphNode* is the basic unit for deriving the parallel strategy. It is a container of operators collectively used together. GraphNode is introduced because *the sharding decision is interrelated within a layer*: a decision on the previous tensor will affect the tensor after. In the GraphNode representation, TAPAS tracks the input/output shape and the split axis of each tensor. For instance, in Figure 3, a dense layer can be a GraphNode, which consists of a matrix multiplication (MatMul) op, an addition (BiasAdd) op, and an activation (ReLU) op that has no weight.

ShardingPattern. A *ShardingPatterns* is a possible parallelised implementation of a GraphNode. For instance, a 2D matrix weight can be split on either dimension or replicated. Therefore, a GraphNode can have multiple valid ShardingPatterns. For each GraphNode, TAPAS defines its sharding patterns using the SRC expression (subsubsection 4.2.1) to separate the definition from implementation. Once the implementations are decided, TAPAS can estimate its cost based on the communication pattern and tensor size.

Parallel strategy. A parallel strategy is a parallelized implementation of the original graph. It is constructed by substituting the GraphNodes with their parallel implementations. The cost of a parallel strategy is calculated as the sum of the costs of all constituting sharding patterns.

4.2.1 Split-Replica-Communication Expression. TAPAS represents all parallel strategies using an abstraction called *Split- Replica- Communication (SRC)*:

Split. *Split* means sharding the tensor on a target axis, after which different devices store different partitions. For example, $S_0(T)$ means sharding tensor T on the first axis. Under this view, data parallelism is just a special case for tensor parallelism where the tensor shards on the batch dimension.

Replica. *Replica* ($R(T)$) means to replicate the tensor T on different devices. For instance, in data parallelism, the weight tensors are replicated while the input tensors are shared.

Communication. Additional communication operators may be needed to combine the partial results. For instance, AllReduce (C_{AR}) is needed in data parallel to aggregate gradients, while AllGather (C_{AG}) is required to exchange partial values after a split operation. It is worth noting that a parallel schedule may not include all three strategies.

Under the SRC expression, an operation

$$Y = Op(A, B)$$

can be expressed using SRC expression as:

$$Y = C(Op(S/R(A), S/R(B))$$

With the SRC expression, we can define general tensor parallel rules for each operator as ShardingPattern. After the best parallel strategy is selected, sharding patterns will be materialized (reconstructed) into a parallelized graph.

The benefits of using SRC is threefold. Firstly, compared to the abstractions in other works [41, 43, 47], SRC reduces the amount of effort necessary to define parallel implementations for new operators/layers. Secondly, having known the tensor shape, SRC can enable symbolic shape checks to validate the parallel strategies. This is crucial because most consecutive sharding patterns are not compatible, and the search can backtrack earlier if a strategy is deemed invalid. Last but not least, SRC can be used to express complex patterns through nested expression.

4.3 Subgraph Mining

Given a GraphNode graph, TAPAS searches for the set of unique subgraphs within it. In order to find significant subgraphs to avoid excessive search, we can control the minimal threshold for subgraph occurrence and subgraph size using *minSupport* and *minSize* respectively.

Algorithm 1 Apriori Frequent Subgraph Search

Require: G(V, E), minSupport, minSize
Ensure: Set of subgraphs with at least minSize nodes
1: Initialize $F \leftarrow$ empty list ▷ Initialize frequent subgraphs
2: Initialize $C \leftarrow$ list of V ▷ Initialize candidate set
3: **for** each subgraph s in C **do**
4: **if** Count(s in G) $\geq$ minSupport **then**
5: Add s to F
6: $C \leftarrow$ subgraphs in F ▷ Candidates for merging
7: **for** $k = 2$ to $|V|$ **do**
8: $C_k \leftarrow$ Merge subgraphs in F_{k-1} that share edges
9: **for** each subgraph s in C_k **do**
10: **if** Count(s in G) $\geq$ minSupport and $|s| \geq$ minSize **then**
11: Add s to F_k ▷ Add subgraph of size k
12: **if** F_k is empty **then**
13: Break ▷ Stop if no more freq. subgraph of size k
14: $C \leftarrow$ subgraphs in F_k
return F

Our subgraph mining algorithm is inspired by the Apriori algorithm in frequent itemset mining. It starts by treating each node as a single-node subgraph and considers them as initial candidates. Each candidate subgraph is then counted in the graph for frequency, and if its frequency surpasses the minimum support threshold and its size meets a minimum number of nodes criterion, it is considered a significant subgraph and added to the output list (line 3-6). It then iteratively expands these candidates by merging the existing frequent subgraphs that share a common edge, thereby generating larger candidate subgraphs (line 8). The algorithm repeats this process until it finds no new frequent subgraphs. The final output is a list of all identified frequent subgraphs that contain at least a specified number of nodes. As `minSupport` is directly linked to the frequency of subgraph, we set to be the number of layers.

4.4 Parallel Strategy Exploration

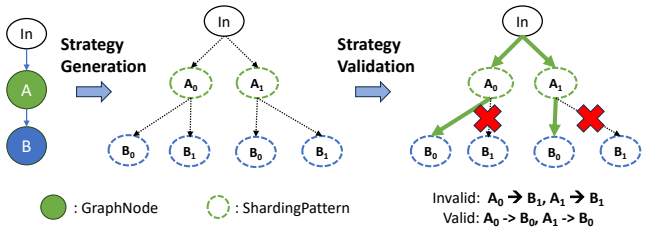

Figure 4: Strategy generation and validation.

After finding the frequent subgraphs, TAPAS moves on to the Strategy Exploration phase, as illustrated in Figure 4. Recall that each subgraph consists of one or more GraphNodes, with each of them potentially having multiple ShardingPatterns. Therefore, this step is accomplished by enumerating all possible combinations of ShardingPatterns within each subgraph and subsequently interlinking them.

However, simply connecting these ShardingPatterns does not guarantee a functional parallel strategy. Because of possible issues such as shape mismatch between subgraphs, not all combinations yield *valid* parallel strategies. To address this, TAPAS employs the SRC to conduct a symbolic shape check on the ShardingPatterns.

The symbolic shape check involves analyzing the shapes of tensors in the computational graph and ensuring that the operations are compatible across ShardingPatterns. The shape propagation is made possible with GraphNodes (which records the shape information of the original tensor) and ShardingPattern (which tracks the transformation rules). As shown in Figure 4, the strategy is valid only if every *pair* of consecutive ShardingPattern is valid; otherwise, it is deemed invalid and we can early stop it without exploring this strategy to the fullest. The compatibility test is particularly crucial when tensors are divided across different devices, and we observe that the vast majority of the strategies are invalid.

Following the validation of strategies, TAPAS constructs the successful parallel strategies using a Breadth-First-Search (BFS) starting from the root node. Subsequently, TAPAS evaluates the performance of each strategy with a cost model and selects the strategy offering the best performance.

4.5 Graph Reconstruction

After strategy validation, the best strategy is chosen from the candidates based on the cost model. TAPAS reconstructs the best strategy back into the computational graph form with the help of the ShardingPatterns. Each ShardingPattern gets materialized into operators by replacing the original subgraphs with the parallelized implementation specified by SRC. After the parallelized graph is ready, it is passed to the training framework backend for execution.

4.6 Communication-Based Cost Model

An accurate cost model is critical to the evaluation of candidate strategies. We profile different tensor parallel schedules of T5-large model using 8 and 16 GPUs (denoted as 8w/16w), and present the result in Figure 5.

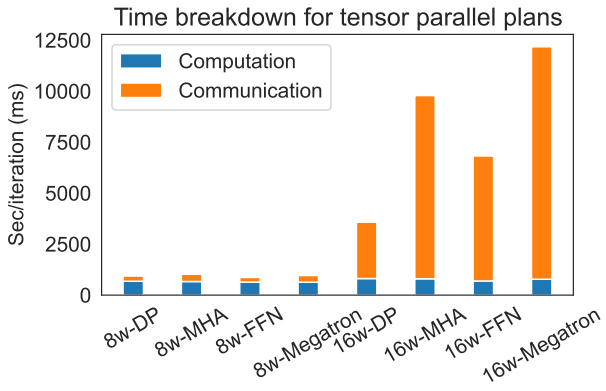

Figure 5: Profiling result for TP schedules of T5-large. *DP*: data parallel, *MHA*: sharding attention only, *FFN*: sharding the feed-forward layer only. *Megatron* refers to the strategy that shards both MHA and FFN.

Each node is equipped with 8 GPUs and interconnected using Ethernet. We observe that inter-node communication is the main bottleneck for tensor parallelism, because the internode interconnect (eg. Ethernet) is usually an order of magnitude slower than the intranode interconnect (eg. PCI-e or NVLink). Therefore, TAPAS employs a communication-based cost model.

Prior works [1, 16, 41, 51] adopt the vanila $\alpha - \beta$ cost model, where α captures the network latency for each message, and β captures the inverse of bandwidth. The total time to send a message of size N is $T(N) = \alpha + \beta N$.

However, this does not account for the communication-computation overlapping scheme in deep learning frameworks. Specifically, we observe that overlapping exists in the backward propagation phase, inside the collective communication operation, and adjacent computation and communication. We therefore propose two changes to the cost model to capture the overlapping effect on communication cost.

Gradient/weight update overlap in backward pass. During the backward propagation phase, gradients are calculated with respect to model's parameter and synchronized across workers. Instead of waiting for the entire gradient computation to finish before starting communication, most DL frameworks implement a gradient overlapping techniques allow the gradient to be communicated as soon as they are computed. Therefore, gradient synchronization on later layers without blocking the backward computation on early layers. We find this optimization technique to reduce the total execution time of backward phase, and use a discount factor γ ($0 < \gamma \leq 1$) to quantify the extent of gradient overlap during the backward pass.

Communication-reduction overlap in collective communications. Collective communication may be overlapped with preceding/succeeding communication operations by decomposition into smaller and data-independent steps [4, 32, 45]. Inside the collective communication operation (eg. AllReduce), the reduction and communication may also be overlapped. TAPAS uses a coefficient ϵ ($0 < \epsilon \leq 1$) to capture the overlapping effect of each collective communication primitive, collected through offline profiling.

Solution. TAPAS addresses these issues using an analytical cost model that treats backward/forward pass separately and different collectives separately. The total cost is the summation of all costs of sharding patterns p found along the computational graph's critical path. The cost of each sharding pattern is determined by the latency and transmission delay, where the latency is linear with respect to the number of participating workers (W), and transmission delay is a factor of the message size in forward pass(N_{fwd}) and backward pass (N_{bwd}), the bandwidth ($1/\beta$), and overlapping factor of collective communication (ϵ).

$$N_p = N_{fwd(p)} + N_{bwd(p)} \tag{1}$$

$$T_{latency(p)} = \alpha' \cdot W \tag{2}$$

$$T_{trans(p)} = \beta \cdot (N_{fwd(p)} + \gamma * N_{bwd(p)}) \cdot \epsilon \tag{3}$$

$$Cost(P_G, S) = \sum_{p=1}^{P} (T_{latency(p)} + T_{trans(p)}) \tag{4}$$

5 Evaluation

We seek to answer the following questions during the evaluation:

- Search speed and training performance: How fast are the *search* time and *throughput* of TAPAS compared with other automatic and manual frameworks?
- Scaling experiments: How well can TAPAS scale on larger models and larger systems?
- Interpretation of discovered strategies: What can be learned from the discovered parallel strategies?
- Micro benchmarks: How robust is the subgraph mining algorithm?

5.1 Evaluation Setup

TAPAS is implemented using 12K lines of Python code on TensorFlow (TF). We assess TAPAS across a diverse set of models, including the dense transformer model (T5), the sparse mixture-of-experts model (GShard MoE), and the convolutional neural network (ResNet). T5 features an encoder-decoder transformer architecture, representing a broad spectrum of large language models like BERT, Llama-1/2/3, GPT-1/2/3, and M6.

We choose Alpa as a evaluation baseline for automatic intra-op parallel framework. For expert-engineered tensor parallel frameworks, we employ Megatron [27] for the T5 model and DeepSpeed [33] for non-transformer models such as ResNet and GShard MoE, in line with the evaluation methods in [8, 41, 51].

Given the diversity of backend frameworks in use, such as TensorFlow, PyTorch, and JAX, we follow the convention in [27, 51] by reporting the performance in FLOPs. This involves calculating the FLOPs for dense matrix multiplication operations per iteration and then dividing this by the iteration time. This method allows for a consistent and comparable evaluation of performance across different frameworks. In cases where evaluations are conducted on

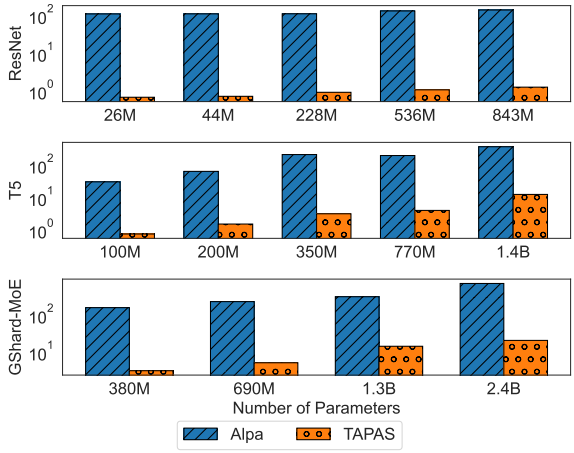

Figure 6: End-to-end search time (in minutes) under different frameworks.

the same backend (between TAPAS and TensorFlow in Figure 8), we report performance metrics directly in terms of iteration time to ensure accuracy and relevancy.

The evaluation was performed on Company A's public cloud nodes. Each node was equipped with 756GB main memory, 2× Intel 8163 CPUs, and 8× Nvidia V100 SXM2 32GB GPUs. The evaluations were performed using FP32 precision. These compute nodes were interconnected by 100 Gbps ethernet.

Each representative model is scaled to various sizes. The T5 model is scaled by adding new layers (depth), the MoE model is scaled by adding experts and layers (width and depth), and the ResNet model is scaled by enlarging the classification layer (width).

5.2 End-to-End Evaluation

In this section, we compare with auto-parallel framework Alpa on both the search time and quality of the discovered strategies.

5.2.1 Search time. Search time is defined by the duration of strategy derivation, excluding framework initialization time and actual training time. In Figure 6, we present the end-to-end search time for increasing model sizes.

To scale the model size along the width, we increase the size of the classification layer of the ResNet model. The base ResNet50 model has 1024 classes in the fully connected (FC) layer. As we increase the dimensions for the FC layer from 1024 to 10K, 100K, 250K, and 400K, the total number of parameters also scales up. As shown in the large-scale classification task with ResNet, TAPAS is two orders of magnitude faster than Alpa in finding the optimal solution, outperforming the latter by $103 - 162\times$.

To scale the model along the depth, we increase the number of transformer layers for T5. Figure 7 shows that, with an increasing number of parameters, TAPAS can still find a plausible schedule in under 15 mins, which is $21 - 67\times$ faster than Alpa.

We further analyze the time breakdown during the search. The efficient graph pruning algorithm greatly shrinks the search space while preserving key optimization space. Moreover, the analytical cost model used by TAPAS does not require operator profiling. As a result, Alpa takes 197 minutes to search 16 candidate strategies, while TAPAS requires only 6 minutes to examine 729 strategies for T5-large.

5.2.2 Training speed. We evaluate the performance of both manual and automatic parallel frameworks, and present them in Figure 7.

ResNet result. Both Alpa and TAPAS excel compared to data parallelism (DP), particularly in handling larger dense layers where Alpa tends to falter. DP duplicates the weight across all workers, quickly exhausting the memory. On the other side, DeepSpeed addresses the memory constraints by sharding the optimizer states and gradients across workers. However, this leads to an increase in the amount and size of messages, particularly for convolutional operators during the backward passes, impacting efficiency. TAPAS shards the fully connected (FC) layers while duplicating the base model, which effectively minimizes the memory burden and reduces the overhead associated with transmitting smaller messages.

T5 result. Data parallelism and Megatron perform better than both Alpa and TAPAS when the model size is less than 760M parameters, while both Alpa and TAPAS outperform them on larger models. This is because Alpa performs reduce-scatter optimization to save communication, while TAPAS uncovers a novel parallel strategy that shards the feed-forward layer while replicating the self-attention layer inside a transformer. The self-attention layer is usually computationally intensive, but the feed-forward layer has two dense matrices. Therefore, sharding only the feed-forward layer can reduce the amount of weight updates while keeping the computational intensity high.

GShard-MoE result. In the GShard-MoE experiments, TAPAS found an expert-level parallel strategy similar to Alpa, but it was discovered using a much smaller search space. The strategy partitions the expert dimension in the MoE layers, while it replicates the weight for non-expert layers like attention and MoE gates. DeepSpeed adopts a similar strategy as the original GShard implementation [21], and combines that with ZeRO-2 DP. However, DeepSpeed falls short when the number of experts does match with the number of worker GPUs. This highlights the need for automatic parallelism.

5.3 Scaling Experiments

We scale TAPAS on ResNet, T5, and MoE models with an increasing number of GPUs while keeping the per-GPU workload constant, and present the result in Figure 8.

Baseline. The baseline is TensorFlow trained with data parallelism. For all models, we first saturate the GPU memory by increasing the batch size until OOM occurs on a single GPU, and linearly scale the batch size with the number of GPUs. The size of the parameters of all base models (on 1 GPU) ranges from 0.77B to 1.3B.

Result. We set a time limit of 120 minutes for the exhaustive search version of TAPAS. It is worth noting that the performance gap between the exhaustive search version of TAPAS (TAPAS-ES) and the subgraph-pruned version(TAPAS-GP) is within 1.5% across the experiments. However, when increasing the size of compute cluster, TAPAS-ES frequently exceeds the time limit due to increased number of parallel strategies. This highlights the challenge of vast search space in auto-parallel frameworks.

We also observe that TAPAS uncovers distinct sharding strategies on different cluster scales for the same model. For MoE (1.3B)

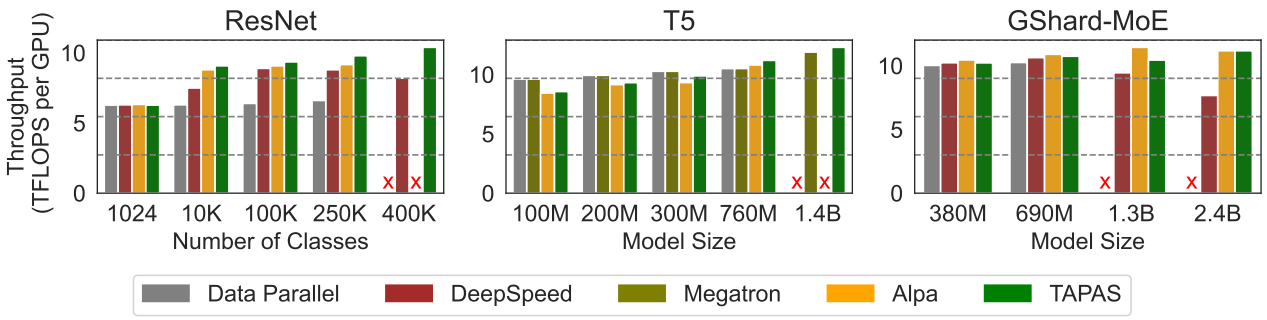

Figure 7: Performance across frameworks on 8 GPUs. "×" represents out-of-memory failure.

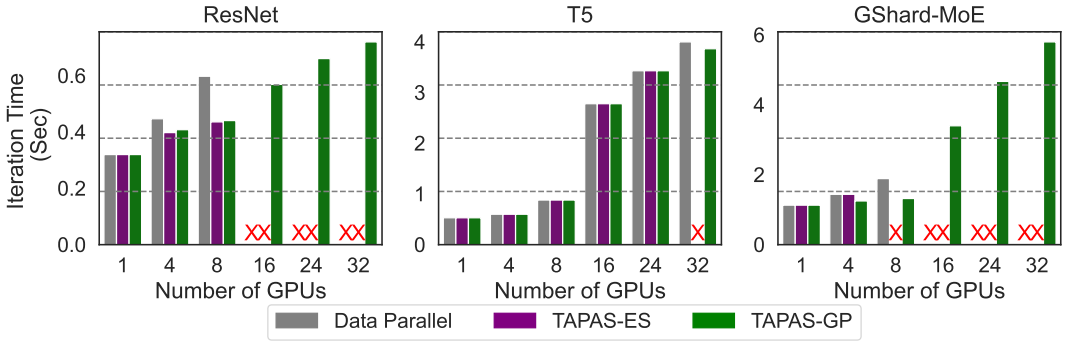

Figure 8: Weak scaling performance. TAPAS-ES: TAPAS with exhaustive search. TAPAS-GP: TAPAS with subgraph pruning.

model, TAPAS discovers that a nested Expert+Tensor Parallel strategy works best by further sharding the feedforward network within an expert layer. This can be helpful when using wide expert layers, which is common for language translation tasks.

On the ResNet (843M) model, we observe that when the size of the classification layer increases, more memory buffers are needed for caching gradients on each worker, hence the DP performance is worse than the TAPAS' schedule from 8 GPUs. After that, OOM error occurs on the DP schedule.

5.4 Visualization of discovered strategies.

We plot the parallel strategies found by TAPAS in Figure 9. Our exploration uncovers that TAPAS is not only capable of identifying fully sharded strategies that mirror Megatron-LM, but also can it unearth novel strategies. These strategies involve partitioning either the multi-head attention (*MHA-only*) or the feed-forward layers (*FFN-only*). Intriguingly, the most effective strategy for dense transformers is the FFN-only plan. This is because FFN is composed of large MatMuls and can still achieve high arithmetic intensity even after splitting, whereas attention module has smaller weights and lower arithmetic intensity. Therefore, when memory ceiling permits, it is better to replicate the attention weight while splitting on the FFN weights. TAPAS also discovers other fully-sharded plans, but they are not selected as they will incur higher communication cost with the same amount of compute reduction. Our result challenges the belief that expert-engineered baseline is universally optimal, and demonstrates the effectiveness of our approach.

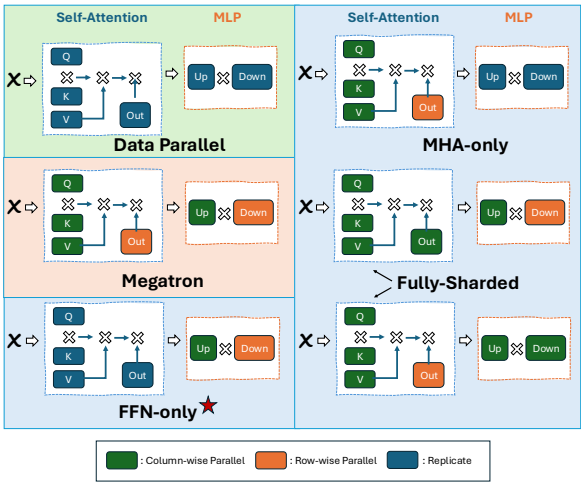

Figure 9: Visualization of selected sharding strategies discovered by TAPAS. LayerNorm/dropout/activation are ignored for presentation clarity.

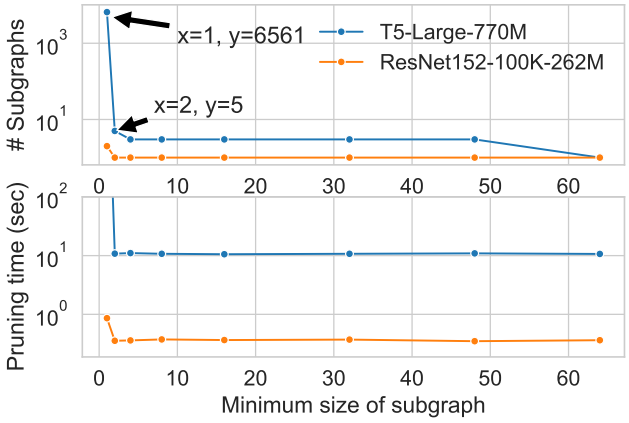

Figure 10: Subgraph pruning evaluations.

5.5 Micro Benchmark

In Algorithm 1, `minSize` determines the threshold for subgraph size. If the threshold is too low, it will still face the exploding search spaces by dealing with too many small graphs; if the threshold is too high, we may see too few subgraphs, resulting in a longer search time or sub-optimal policy. Since the architecture of different neural networks may vary significantly, it is desirable to have a robust threshold that does not require extensive tuning.

We explore a range of `minSize` and report the number of unique subgraphs found and subgraph mining algorithm running time in Figure 10. Take the T5-Large model with 770M parameters as an example: when the threshold is 1, meaning the graph is kept unfolded, it contains 6561 nodes. After subgraph mining, the number of unique subgraphs has drastically been reduced to just 5. As the threshold changes, the number of identified unique subgraphs stays relatively stable, showing that our algorithm is robust on different model architecture.

Furthermore, we observe that the subgraph mining algorithm is efficient, taking less than 12 seconds to find the subgraphs for T5-large, less than a second for the 152-layer 100K-class ResNet model, and 30 seconds for the 1.3B MoE network. This signals that TAPAS can scale well on large foundation models.

5.6 Limitation and Future Work

To extend TAPAS to pipeline parallel strategy, we can update the subgraph selection algorithm by choosing the sub-computation graphs as pipeline stages while satisfying load balancing constraints across subgraphs.

To further optimize the memory consumption, TAPAS could leverage other orthogonal techniques such as mixed precision [25], gradient recomputation [6, 18]. Also, gradient checkpointing can be used to offload the selected GraphNode onto the main memory.

It is worth noting that our approach may share some similarities with the pattern-matching technique used in TVM [5] for operator fusion. The key difference is TAPAS targets the training setting, which is more challenging due to having dynamic tensor shape and unknown subgraph pattern. TVM mainly targets the inference setting, where the patterns are pre-defined in the ML compilers based on expert experience, and the tensor shapes are known.

6 Conclusion

We present TAPAS, an automatic parallelism framework that efficiently discovers tensor parallel plans for large neural networks. Leveraging the observation that shared subgraphs widely exist in neural networks, we design TAPAS, an automatic parallel framework that significantly reduces redundant search effort by subgraph mining and early stopping. We also built an analytical cost model that accurately captures the amount of communication during tensor parallel training. The best parallel strategies discovered by TAPAS not only measure up to expertly engineered strategies, but also excel in search speed, reducing the strategy derivation time by two orders of magnitude compared to the state-of-the-art system. In summary, TAPAS provides a scalable, fast, and automatic solution for tensor parallelism that can help alleviate the burden of manual tuning.

Acknowledgments

This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 1 (T1 251RES2409).

References

[1] Behnaz Arzani, Siva Kesava Reddy Kakarla, Miguel Castro, Srikanth Kandula, Saeed Maleki, and Luke Marshall. 2023. Rethinking Machine Learning Collective Communication as a Multi-Commodity Flow Problem. *arXiv preprint arXiv:2305.13479* (2023).

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

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

[4] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. 2024. Centauri: Enabling Efficient Scheduling for Communication-Computation Overlap in Large Model Training via Communication Partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 178–191.

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

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

[7] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[8] Weihao Cui, Zhenhua Han, Lingji Ouyang, Yichuan Wang, Ningxin Zheng, Lingxiao Ma, Yuqing Yang, Fan Yang, Jilong Xue, Lili Qiu, et al. 2023. Optimizing dynamic neural networks with brainstorm. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 797–815.

[9] Jacob Devlin, Ming Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. *BERT: Pre-training of deep bidirectional transformers for language understanding*. Technical Report. 4171–4186 pages. https://github.com/tensorflow/tensor2tensor

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In

9th International Conference on Learning Representations (ICLR 2021). https://openreview.net/forum?id=YicbFdNTTy

[11] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: A pipelined data parallel approach for training large models. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP* 21 (2021), 431–445. doi:10.1145/3437801.3441593

[12] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. 2022. FasterMoE: Modeling and Optimizing Training of Large-Scale Dynamic Pre-Trained Models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 120–134. doi:10.1145/3503221.3508418

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Vol. 2016-Decem. 770–778. doi:10.1109/CVPR.2016.90

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, Hyouk Joong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in Neural Information Processing Systems* 32 (2019).

[15] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. 2022. Whale: Efficient Giant Model Training over Heterogeneous GPUs. In *USENIX Annual Technical Conference*. USENIX.

[16] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of the 2nd Conference on Machine Learning and Systems (MLSys)*. https://proceedings.mlsys.org/paper/2019/file/78530480f14bc7b2879ae05070c78c11-Paper.pdf

[17] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. (2020). http://arxiv.org/abs/2001.08361

[18] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2021. Dynamic Tensor Rematerialization. In *Proceedings of the 9th International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=Yl2aDBJRTm Spotlight paper.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

[20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[21] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *9th International Conference on Learning Representations (ICLR 2021)*. https://openreview.net/forum?id=qrwe7XHTmYb

[22] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. 2021. TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models. In *Proceedings of the 38th International Conference on Machine Learning (ICML 2021)*. 6543–6552. https://proceedings.mlr.press/v139/li21d.html

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

[24] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2022. HET: Scaling out Huge Embedding Model Training via Cache-enabled Distributed Framework. In *Proceedings of the VLDB Endowment (PVLDB)*, Vol. 15. 312–320. https://www.vldb.org/pvldb/vol15.html

[25] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hooman Wu. 2018. Mixed Precision Training. *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)* (2018).

[26] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. Pipedream: Generalized pipeline parallelism for DNN training. *SOSP 2019 - Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), 1–15. doi:10.1145/3341301.3359646

[27] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* (2021). doi:10.1145/3458817.3476209

[28] Xiaonan Nie, Xupeng Miao, Zilong Wang, Zichao Yang, Jilong Xue, Lingxiao Ma, Gang Cao, and Bin Cui. 2023. FlexMoE: Scaling Large-scale Sparse Pre-trained Model Training via Dynamic Device Placement. In *Proceedings of the 2023 ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD)*. 110:1–110:19. https://dl.acm.org/doi/10.1145/3588964.3591467

[29] Penghui Qi, Xinyi Wan, Nyamdavaa Amar, and Min Lin. 2024. Pipeline Parallelism with Controllable Memory. *arXiv preprint arXiv:2405.15362* (2024).

[30] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. *Exploring the limits of transfer learning with a unified text-to-text transformer*. Technical Report. 1–67 pages. http://jmlr.org/papers/v21/20-074.html.

[31] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* 2020-Novem (2020). doi:10.1109/SC41405.2020.00024

[32] Saeed Rashidi, Matthew Denton, Srinivas Sridharan, Sudarshan Srinivasan, Amoghavarsha Suresh, Jade Nie, and Tushar Krishna. 2021. Enabling compute-communication overlap in distributed deep learning training platforms. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 540–553.

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

[34] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-offload: Democratizing billion-scale model training. *2021 USENIX Annual Technical Conference* (2021), 551–564. https://www.deepspeed.ai/tutorials/

[35] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*. Springer, 234–241.

[36] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Neural Information Processing Systems*.

[37] Ziji Shi, Jialin Li, and Yang You. 2024. ParaGAN: A Scalable Distributed Training Framework for Generative Adversarial Networks. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*.

[38] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC19)*. https://dl.acm.org/doi/10.1145/3295500.3356143

[39] Yi Tay, Mostafa Dehghani, Jinfeng Rao, William Fedus, Samira Abnar, Hyung Won Chung, Sharan Narang, Dani Yogatama, Ashish Vaswani, and Donald Metzler. 2022. Scale Efficiently: Insights from Pre-Training and Fine-Tuning Transformers. In *10th International Conference on Learning Representations (ICLR 2022)*. https://openreview.net/forum?id=aNiqNrhNzwb

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

[41] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain, Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat Mccormick, Jamaludin Mohd-yusof, Jongsoo Park, Misha Smelyanskiy, Alex Aiken, Pat Mccormick, Jamaludin Mohd-yusof Xi, and Luo Dheevatsa. 2022. Unity : Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization This paper is included in the Proceedings of the. (2022). https://www.usenix.org/conference/osdi22/presentation/unger

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* 2017-Decem (2017), 5999–6009.

[43] Minjie Wang, Chien chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. *Proceedings of the 14th EuroSys Conference 2019* (2019). doi:10.1145/3302424.3303953

[44] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. Distributed GNN Training with Hybrid Dependencies Processing. In *Proceedings of the 2022 ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD)*. 1061–1073. https://dl.acm.org/doi/10.1145/3514221.3517836

[45] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. 2022. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 93–106.

[46] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al.

2022. Emergent abilities of large language models. *Transactions on Machine Learning Research (TMLR)* (2022). https://openreview.net/forum?id=yzkSUxE1e2

[47] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. 2021. GSPMD: General and Scalable Parallelization for ML Computation Graphs. (2021). http://arxiv.org/abs/2105.04663

[48] Fuzhao Xue, Ziji Shi, Futao Wei, Yuxuan Lou, Yong Liu, and Yang You. 2022. Go wider instead of deeper. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 8779–8787.

[49] Hailin Zhang, Zirui Liu, Boxuan Chen, Yikai Zhao, Tong Zhao, Tong Yang, and Bin Cui. 2024. CAFE: Towards Compact, Adaptive, and Fast Embedding for Large-scale Recommendation Models. In *Proceedings of the 2024 ACM International Conference on Management of Data (SIGMOD)*. https://dl.acm.org/doi/10.1145/3639306

[50] Yuhao Zhang and Arun Kumar. 2023. Lotan: Bridging the Gap between GNNs and Scalable Graph Analytics Engines. *Proceedings of the VLDB Endowment* 16, 3 (2023), 312–324. https://www.vldb.org/pvldb/vol16.html

[51] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*. 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng

A Complexity Analysis of Existing Works

Following the discussion on related work, we analyze the complexities of two other automatic model parallel frameworks and present it in Table 1. We define the total complexity as:

$$total\_complexity = search\_complexity + num\_plans * evaluation\_complexity$$

A.0.1 FlexFlow. FlexFlow operates on four dimensions (Sample, Operator, Attribute, and Parameter), and there was no space reduction. Therefore, the search space is $N(4E, 4V)$. As search complexity, FlexFlow employs the Markov chain Monte Carlo (MCMC) algorithm. Thus, we use B to denote the computational budget (number of trials) in MCMC sampling. Furthermore, within each trial, it needs to evaluate its performance by querying the cost model with Depth-First-Search(DFS), hence its evaluation complexity is $O(V + E)$.

A.0.2 Alpa. Alpa is formulated as a multi-level optimization problem: in the outer loop, it searches for the inter-op parallel plan using dynamic programming; in the inner loop, it finds the intra-op parallel plan using integer linear programming. First, since it operates at MLIR HLO, which is a finer IR than the TensorFlow operator, we formulate the search space as $N(kE, kV)$ where $k \geq 1$. In the outer loops, it uses a similar algorithm to [22] to search for pipeline slices and map the slices to devise mesh. Optimization like operator clustering and early pruning reduces the outer loop complexity to $(kV)^2L$. For the inner loop, since the exact complexity of their ILP solver is unknown, we use a lower bound by performing a BFS from each operator, and the complexity is given as $kE(kV + kE)$. Finally, each trial needs to evaluate its performance by querying the cost model, so the evaluation complexity is $kV + kE$.

A.0.3 TAPAS. In TAPAS, we first reduce the search space by converting the TensorFlow graph to TAPAS graph (by $C\times$, where $C \geq 1$). We then prune the tree by layer, further reducing the complexity to $N(\frac{E}{2CL}, \frac{V}{2CL})$. In the searching stage, the result is derived by performing a BFS. Thus, the complexity is $\frac{V+E}{2CL}$. For the evaluation stage, TAPAS needs to evaluate the cost of each plan by querying the cost model, which depends only on the size of the edges. Thus, the evaluation cost is $\frac{E}{2CL}$.

Table 2: Ablation study of cost model optimizations. CF: constant filter, GO: Gradient Overlapping, EC: Efficiency of Collective Communications.

Baseline	CF	GO	EC	Acc@1	Acc@5	MRR
✓				0.53	0.86	0.71
✓	✓			0.53	0.93	0.68
✓	✓	✓		0.73	1	0.84
✓	✓	✓	✓	0.87	1	0.92

B Ablation Study on Cost Model Optimizations

TAPAS cost model outputs a score for each candidate strategy and ranks them based on the score. We examine both the accuracy (whether the best strategy returned by the cost model is indeed the best strategy during runtime) and the order (how far off is the best strategy in the ranking). In this section, we study the effectiveness of optimizations on the cost model on 15 different architectures (5× T5, 6× CNN, and 4× MoE model).

For top-K candidates, we use the following metrics for evaluation:

- Accuracy@K: the probability of the best-performing strategy being found within top-K of the cost model ranking; and
- Mean Reciprocal Ranking(MRR): the correctness of the ranking weighted by the relative position of the best strategy in ranking, defined as:

$$MRR = \frac{1}{Q}\sum_{i=1}^{Q}\frac{1}{\text{rank}_i}$$

  where rank_i is the rank of the ground truth best strategy in the cost model ranking.

MRR captures the relative order of strategies by encouraging better strategies placed at higher ranks. A higher MRR indicates that the best-performing strategy is ranked higher. Since we have 15 model architectures, $|Q| = 15$.

From Table 2, it's clear that the performance enhancements in our cost model primarily stem from communication overlap and the efficiency of collective communication primitives. The former optimization addresses non-overlapping communications, while the latter takes into account efficiency variances among primitives, resulting in better alignment with real training performance. The act of filtering constant tensors, while it may not yield significant improvements independently, is indeed a necessary step in the process.