ACESO: Efficient Parallel DNN Training through Iterative Bottleneck Alleviation

Guodong Liu*
State Key Lab of Processors, Institute of Computing Technology, CAS
Univ. of Chinese Academy of Sciences
liuguodong19z@ict.ac.cn

Youshan Miao
Microsoft Research
yomia@microsoft.com

Zhiqi Lin
University of Science and Technology of China
zhiqi.0@mail.ustc.edu.cn

Xiaoxiang Shi
Shanghai Jiao Tong University
lambda7shi@sjtu.edu.cn

Saeed Maleki
Microsoft Research
saemal@microsoft.com

Fan Yang
Microsoft Research
fanyang@microsoft.com

Yungang Bao
State Key Lab of Processors, Institute of Computing Technology, CAS
Univ. of Chinese Academy of Sciences
baoyg@ict.ac.cn

Sa Wang
State Key Lab of Processors, Institute of Computing Technology, CAS
Univ. of Chinese Academy of Sciences
wangsa@ict.ac.cn

Abstract

Many parallel mechanisms, including data parallelism, tensor parallelism, and pipeline parallelism, have been proposed and combined together to support training increasingly large deep neural networks (DNN) on massive GPU devices. Given a DNN model and GPU cluster, finding the optimal configuration by combining these parallelism mechanisms is an NP-hard problem. Widely adopted mathematical programming approaches have been proposed to search in a configuration subspace, but they are still too costly when scaling to large models over numerous devices.

ACESO is a scalable parallel-mechanism auto-configuring system that operates iteratively. For a given parallel configuration, ACESO identifies a performance bottleneck and then, by summarizing all possible configuration adjustments with their resource consumption changes, infers their performance impacts to the bottleneck and selects one that mitigates the bottleneck. This process repeats for many iterations until a desired final configuration is found. Unlike mathematical programming approaches that examine the configurations subspace to find the optimal solution, ACESO searches in the configuration space in a stochastic approach by repeatedly identifying and alleviating bottlenecks. ACESO significantly reduces configuration searching cost by taking the approach of resolving one bottleneck at a time. This allows ACESO to find configurations that would be usually missed in subspace search approaches. We implemented and tested ACESO on representative DNN models. Evaluations show that it can scale to 1K-layer models. Compared to state-of-the-art systems, ACESO achieves up to 1.33× throughput improvement with less than 5% of the searching cost.

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**; **Machine learning**.

Keywords: deep learning, distributed system, automatic parallelization

ACM Reference Format:
Guodong Liu, Youshan Miao, Zhiqi Lin, Xiaoxiang Shi, Saeed Maleki, Fan Yang, Yungang Bao, and Sa Wang. 2024. ACESO: Efficient Parallel DNN Training through Iterative Bottleneck Alleviation. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3627703.3629554

1 Introduction

Large-scale deep learning has recently demonstrated significant success in diverse domains, notably computer vision [13, 25, 26, 50] and natural language processing [3, 7, 38–40]. Recently, OpenAI's ChatGPT has garnered considerable attention due to its remarkable performance, matching human-level capabilities across multiple tasks. Training such a large-scale model requires parallel computation performed over many expensive devices, e.g., GPUs, for a long duration [7, 13, 34, 39, 43], which makes execution efficiency critical. To exploit the efficiency of parallel deep

*This work was done when Guodong Liu was with Microsoft Research (Asia) as an intern from March 2021 to September 2022.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629554

这篇工作最大的问题在于，他宣称 Alpa 这类全空间搜索是 NP-hard 所以时间开销很大，但其实 Alpa 的时间开销主要来自于大量的 profiling 而非搜索算法，但这篇工作实际上也需要 profile 每个 op 在不同划分下的性能 (3.3 节)，所以在 5.3 中可以看到其 profiling overhead 也是很大的

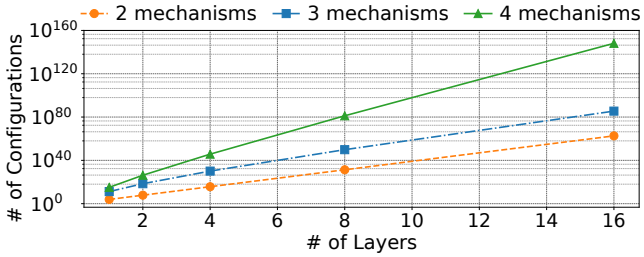


Figure 1. Number of possible configurations (in log scale) increases with the number of model layers and mechanisms. (GPT models on 16 devices, 2 mechanisms include data and tensor parallelism, 3 mechanisms further include pipeline-parallelism, and 4 mechanisms further include recomputation.)

learning execution, many mechanisms, including data parallelism [6, 41], tensor parallelism [18, 43, 48] and pipeline parallelism [15, 28], have been proposed to distribute deep learning task loads among devices, as well as mechanisms that balance the usage of different resources, such as recomputation [5, 12, 16, 20–22] which reduces memory footprint with extra computation.

In parallel deep learning, part of the DNN model may contain multiple different mechanisms, with each mechanism having a specific configuration. For instance, tensor parallelism splits the parameters of an operator among devices, with the configuration: "how many sub-operators it partitions into?". To improve performance in parallel deep learning, multiple mechanisms are combined to work cooperatively [10, 18, 43, 48, 49, 51] on the same model. Therefore, the number of total possible configurations increases exponentially with the number of operators and the number of mechanisms, as shown in Figure 1. As a result, the configuration search time increases accordingly. In fact, searching for optimal parallelization configuration is NP-hard [18, 51], and it is nearly impossible to find an optimal configuration for recent DNN models with one thousand layers [23, 47].

To find an efficient configuration, many algorithms have been proposed [10, 18, 28, 44, 48, 51] to automate the parallelization. However, due to the huge configuration space and the NP-hard complexity, existing parallel systems either search a limited configuration subspace that focuses on only a few mechanisms (with the rest manually configured) or explore the model at a coarse granularity (configuring operator groups instead of configuring each operator individually). For example, tofu [48] and FlexFlow [18] only consider data and tensor parallelisms. PipeDream and Dapple [10, 28] search include data and pipeline parallelisms. DTR [20] focuses on recomputation [5] search. Both Piper [44] and Alpa [51] include data, tensor, and pipeline parallelisms. However, Alpa skips recomputation search, and Piper coarsens the search on layer-level other than operator-level as in other systems. Such reduction of the search space would result in limited performance potential. Nevertheless, even with the space reduction, current works still show non-trivial overhead in configuration search. For example, as shown in our experiment (Exp#2), Alpa spent 3 hours searching for configurations of GPT-3 (13B). Search overhead can be a huge burden when quick reconfiguration is needed, e.g., in a shared cluster with frequent changes in resources.

In this paper, we ask the question: *Can we find an efficient way to explore parallel DNN configurations while considering all of the above mechanisms?*

We found it to be possible if we reconsidered the problem from a different angle: *bottleneck alleviation*. In parallel deep learning, performance is usually constrained by hardware resource ***bottlenecks*** on one or several devices, such as devices in a pipeline stage becoming saturated with computational tasks or reaching their memory capacity limits, which in turn restricts the overall system throughput. Inspired by computation pipeline analysis in traditional parallel programming [30], we can improve the performance of parallel DNN training by identifying bottlenecks and alleviating them with *reconfiguring* mechanisms accordingly. Eliminating a bottleneck may help improve system performance but can also bring about a new bottleneck. We can further improve performance with another round of reconfiguration and iteratively perform reconfigurations to continually improve performance. Such a bottleneck usually only involves a small scope of the entire model, so the search space is significantly reduced.

However, it is still costly to evaluate all possible reconfigurations of different mechanisms that could alleviate bottlenecks in each step. We can further narrow down the search space by taking on a resource perspective and using ***resource trading***. We observed that reconfigurations of various mechanisms can all be considered as *trading* consumption of different hardware resources. For example, increasing data-parallel concurrency (2 GPU data-parallel ⇒ 4 GPU data-parallel) can be viewed as leveraging computation power and memory capacity from more devices but with the cost of extra communication overhead for synchronization; applying recomputation [20] can be considered as trading duplicated computation for less memory footprint. In Table 1, we summarize a set of basic reconfigurations as ***reconfiguration primitives*** (*primitives* for short) and their impacts on the consumption of three types of resources: computation, communication, and memory. We can query the table for eligible reconfigurations when we want to alleviate a bottleneck. For example, if a bottleneck device is computational and memory intensive and communication underutilized, we can query primitives with computation/commutation/memory utilization "↘, < *any* >, ↘" and find that one of the eligible primitives is "inc-tp" — increasing concurrency with tensor parallelism. Applying these reconfiguration primitives reduces the consumption of scarce resources (computation

Bottleneck 是单或多 devices 级别的

需要重配置带来的额外硬件从哪里来？如果是额外的，那这样的迭代岂不是会无限重复下去？因为只要额外分配资源都会提升性能，且这样会错过很多固定资源量下的不同并行策略

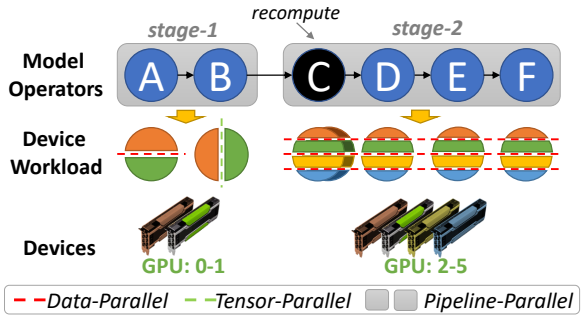


Figure 2. An example of parallel training configuration.

and memory) and leads to higher throughput due to better utilization of the spare resources (communication). It, therefore, allows us to unify the consideration of different reconfiguration primitives in a resource consumption "trading" view and effectively reduces search cost by pruning ineligible primitives.

Our work. Following these observations, we propose Aceso, a simple and scalable parallel configuration searching system. Starting from an initial configuration, Aceso iteratively pinpoints a resource bottleneck, searches the reconfiguration primitives table, and applies the selected primitives to alleviate the bottleneck and improve performance. The algorithm continues this iterative process of "finding and applying" reconfigurations until it reaches convergence (meaning no further reconfiguration can improve performance) or until it exhausts the allotted search time budget. To search efficiently, Aceso introduces an accurate performance model to quickly evaluate configurations and identify performance bottlenecks.

We prototyped Aceso and evaluated it using different representative DNN models. To the best of our knowledge, Aceso is the first system to scale parallelization tuning to 1K-layer models. Our preliminary results show that Aceso can effectively find high-performance configurations for real-world deep learning workloads. Compared with state-of-the-art automated parallel systems like Alpa [51], Aceso takes only less than 5% searching time of Alpa, but the final selected configuration outperforms that of Alpa with 1.33× speedup due to its ability to holistically consider more mechanisms in a unified space. We have open-sourced Aceso at: https://github.com/microsoft/SuperScaler/tree/Aceso.

2 Background and Motivation

2.1 Mechanisms for Parallel Deep Learning

Deep neural network (DNN) training iteratively performs computational operators over mini-batch data (sample or activation) tensors on accelerating devices like GPUs to update model parameter (weight) tensors [41]. For better accuracy, DNN models increase the computation complexity of operators, as well as the memory footprint of weights and activation [2], surpassing the computation power and memory capacity of a single device. Therefore, many parallel training mechanisms, including parallelisms [18, 43, 48, 49] and resource optimizations [5, 22] have been proposed to efficiently distribute the DNN workloads among devices.

Data parallelism. Data parallelism replicates operators and parameter tensors among devices to parallel process partitioned mini-batch training data, in the cost of communication at the end of each iteration as parameter synchronization.

Tensor Parallelism. Tensor parallelism [18, 43, 48] splits weight tensors among devices to support large parameter models, in the cost of activation tensor communication.

Pipeline Parallelism. Pipeline parallelism [10, 15] groups model operators into consecutive stages, with each stage assigned to a device, and connected with simple send-recv communication. To improve hardware utilization, each mini-batch is further partitioned into microbatches. It employs a carefully designed scheduling algorithm to coordinate the operator execution among different microbatches [15, 28].

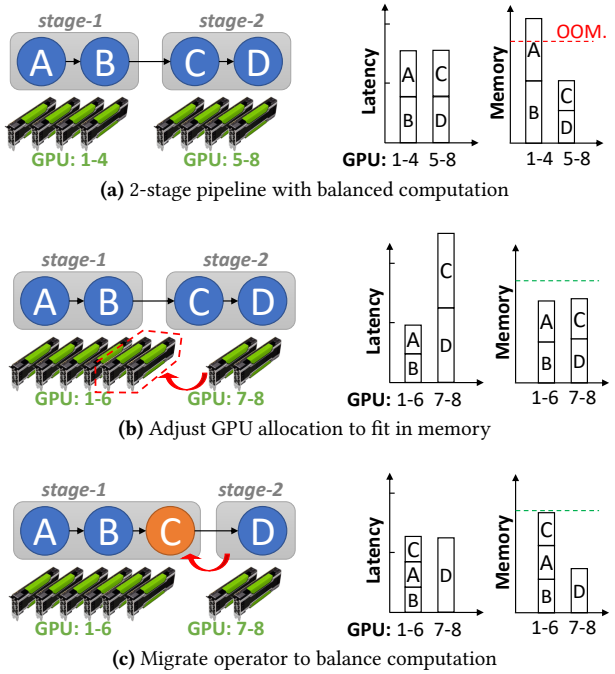
Recomputation. To conserve memory, instead of keeping an operator's computation output for later gradient computing, recomputation mechanisms [5, 20] release the output tensor and "restore" it later for gradient computing with an extra operator computation.

Mechanism Combinations. To harness the advantages of different mechanisms, current solutions [43, 44, 49] hierarchically combine data/tensor/pipeline parallelisms and recomputation for distributed execution. Figure 2 shows a hierarchical combination that first splits the model computation into stages using pipeline parallelism. Each stage is then assigned to a device group. Within each device group, the stage is further parallelized using data/tensor parallelisms or a hybrid combination of these techniques. When the recomputation option is turned on, all the operators in each stage are recomputed to save memory consumption [20].

2.2 Complex Mechanisms Configuring

Each mechanism contains multiple configurable nobs to specify the execution precisely. Configuring each mechanism in a combined solution leads to a huge configuration space, and finding the optimal solution is an NP-hard problem [18, 51]. State-of-the-art works can be divided into two groups: *rule-based solutions* and *automated search solutions*.

Rule-based Solutions. Rule-based solutions pre-define a policy for certain models to quickly provide a good (but not optimal) configuration [21, 29, 42, 43]. For example, Megatron-LM [29] first applies tensor parallelism to GPUs inside a server, then applies pipeline parallelism to a minimum number of servers to fit the model into memory. These "tensor-pipeline-parallelism" groups are then replicated to achieve data parallelism. Because it is designed for a specific model, this approach may not be versatile enough to accommodate other scenarios.



(a) 2-stage pipeline with balanced computation

(b) Adjust GPU allocation to fit in memory

(c) Migrate operator to balance computation

Figure 3. Example bottleneck and alleviation. Right bars show latency and memory consumption on different GPUs.

Automated Search Solutions. For general models and hardware, some systems [17, 18, 44, 48, 51] automate the configuration of parallel mechanisms using mathematical programming, e.g., dynamic programming and integer linear programming. However, searching for the optimal configuration is NP-hard [18, 51]. To reduce search space and make the global problem more tractable, these systems simplify the problem using different methods. Some focus on limited mechanisms, such as only using recomputation [22], data/tensor parallelisms [18, 48], or data/pipeline parallelisms [15, 28], or employing all mechanisms except recomputation [51]. Some other approaches reduce the number of operators under consideration, such as grouping multiple operators and treating them as a single operator during configuration searching [44, 51]. However, simplifying the problem space may lead to the omission of some potentially good configurations, and mathematical programming-based approaches may fall short in complex models with many mechanisms due to the exponential increase in complexity.

2.3 A New Perspective: Bottleneck Alleviation

Bottleneck alleviation is a well-recognized optimization technique in stream processing [19, 37], where pipeline parallelism is also a common parallel mechanism. Existing works [30] have observed that a common system performance bottleneck is often found in the most heavily loaded pipeline stage. Therefore, better system performance can be achieved by alleviating this bottleneck through re-balancing the load among stages.

Such a *bottleneck* is also very common in parallel DNN training but is more complex due to various mechanisms and the unique nature of DNN training. For example, Figure 3(a) shows a sequential model with 4 identical operators. When the computation is divided evenly into 2 stages, out-of-memory issues arise as DNN needs to keep activation, with stage-1 keeping more than stage-2 [28]. It's possible to alleviate such a bottleneck stage with a mechanism reconfiguration. For example, one solution is to assign more GPUs to stage-1, as shown in Figure 3(b). However, the computation time of stage-2 then becomes a bottleneck due to reduced computational power. Therefore, Figure 3(c) considers the bottleneck stage that suffers from high computation cost, migrating operators from stage-2 to stage-1 makes the pipeline stage more balanced. The performance can thereby be improved due to better resource utilization. This shows that improving DNN training through bottleneck alleviation is not only possible but also effective in both locating bottlenecks and generating the corresponding reconfigurations. In this paper, we summarize existing DNN training mechanisms that can potentially help resolve bottlenecks, and leverage the idea of bottleneck alleviation to efficiently achieve high-performance parallel training.

把 stage 2 的资源分一部分给 stage 1

把 stage 2 的一个 op 移给 stage 1

This approach alleviates bottlenecks iteratively, transforming the parallelization configuration search from an NP-hard problem into an SGD-like problem. In addition, it improves efficiency by eliminating reconfigurations on non-bottlenecks of the models and filtering out ineffective reconfigurations. It then enables the discovery of better plans with additional configurations, such as recomputation and in-stage heterogeneous partitioning, as validated in our experiments (§5.4).

3 Aceso Design

We propose Aceso, an efficient and effective system that automates the configuration search of parallel DNN training through iterative bottleneck alleviation. Given a DNN model and hardware devices, Aceso can efficiently find a configuration that precisely defines a high-performance parallel DNN execution.

System architecture and workflow. As shown in Figure 4, Aceso first constructs a performance model for a given model and hardware (§3.3). The performance model is used to speed up the search by quickly predicting a configuration's execution time and resource consumption. Then, Aceso generates an initial configuration and passes it to the iterative reconfiguration search. When the search runs out of time budget, the best configuration is passed to Aceso runtime to perform parallel training accordingly.

Search algorithm overview. Aceso utilizes the bottleneck alleviation idea to guide reconfigurations. As shown in Figure 4, during each iteration of search, Aceso first predicts

整个系统的 bottleneck 位于负载最重的那个 pipeline stage

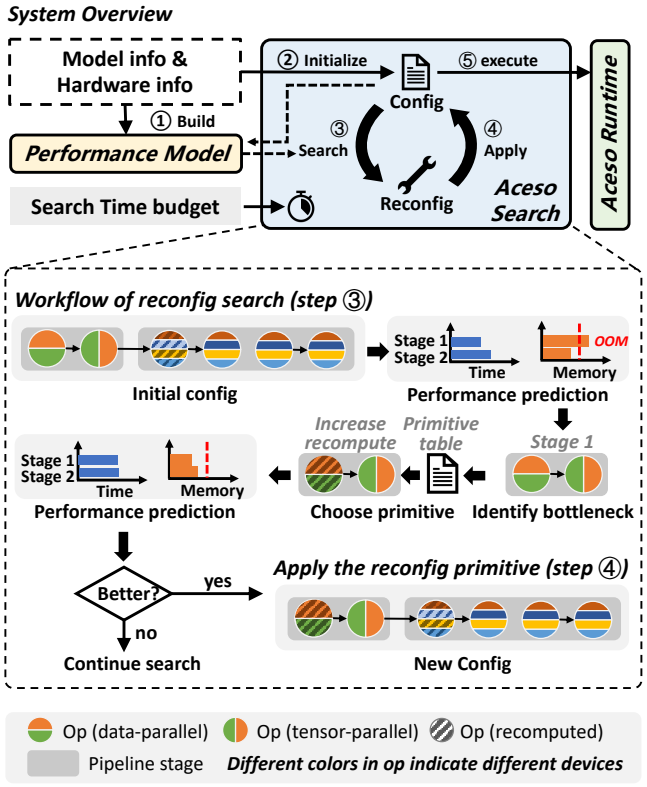


Figure 4. Aceso overview.

the performance of a configuration, then identifies the bottleneck of the configuration. For the identified bottleneck, Aceso chooses the most promising *Reconfiguration Primitive* to alleviate the bottleneck. If one primitive cannot improve the configuration, Aceso continues to search for a sequence of primitives, until it finds a combination of primitives that can lead to a better configuration. (Here, "better" refers to when an infeasible configuration becomes feasible, or when a feasible configuration becomes even faster.) We provide a pseudo-code of the Aceso search algorithm in the Appendix.

At a high level, Aceso needs to answer two key questions during the configuration search:

- Where is the bottleneck? (§3.1)
- How to alleviate the bottleneck for *better* configuration? (§3.2)

3.1 Where is the bottleneck

Before discussing the bottleneck, we will first describe the configuration in Aceso's search space and discuss what is considered a bottleneck in such a configuration. Finally, we will introduce how to identify the top-priority bottlenecks.

Configuration representation. Aceso combines multiple parallel DNN training mechanisms, including data parallelism, tensor parallelism, pipeline parallelism, and recomputation. Given a DNN model over certain hardware devices, Aceso needs to unambiguously express a concrete parallelization strategy that defines each mechanism's setting, called a parallel DNN ***configuration***. Figure 2 shows an example of parallelization configuration. The 6-operator model applies 2-stage pipeline parallelism on the high level, with different tensor/data parallelisms at each stage. This type of representation is compatible with expressing configurations in existing systems such as Megatron-LM and Alpa.

Bottlenecks. Aceso considers bottlenecks in the context of pipeline parallelism, where the bottleneck is the pipeline stage that dominates system performance. This is natural because different pipeline stages may have diverse loads, and bottlenecks usually arise from an imbalance of stages [30]. Even in a balanced pipeline with identical model partitions, earlier stages may consume more memory than the later ones to maintain intermediate states [24]. Contrary to pipeline parallelism, data and tensor parallelisms usually do not introduce such imbalance, and workloads are distributed evenly in a parallelism group. Therefore, when pipeline parallelism is combined with data/tensor parallelisms, devices in the same pipeline stage usually have the same loads. Due to this in-stage symmetry, we can use one GPU from each stage for resource utilization analysis and bottleneck identification. By identifying bottlenecks, we reduce the adjustment targets from any part of the model into one of the stages, significantly reducing the number of decisions that need to be considered.

Importance of bottlenecks. Different resource bottlenecks can lead to different levels of consequences. Out-of-memory (OOM) crashes execution, while a heavy computation or communication load leads to longer execution time but does not break the execution. It is therefore natural to prioritize bottlenecks that may crash the execution to first ensure configuration viability. Aceso leverages its performance model (§3.3) to obtain the estimated computation/communication time and memory consumption when identifying top-priority bottlenecks. In Aceso, we use the following heuristic to identify bottlenecks as future targets for reconfiguration:

Heuristic-1. *When the configuration is out-of-memory, the stage with the largest memory consumption is the bottleneck. Otherwise, the stage with the longest execution time is the bottleneck.*

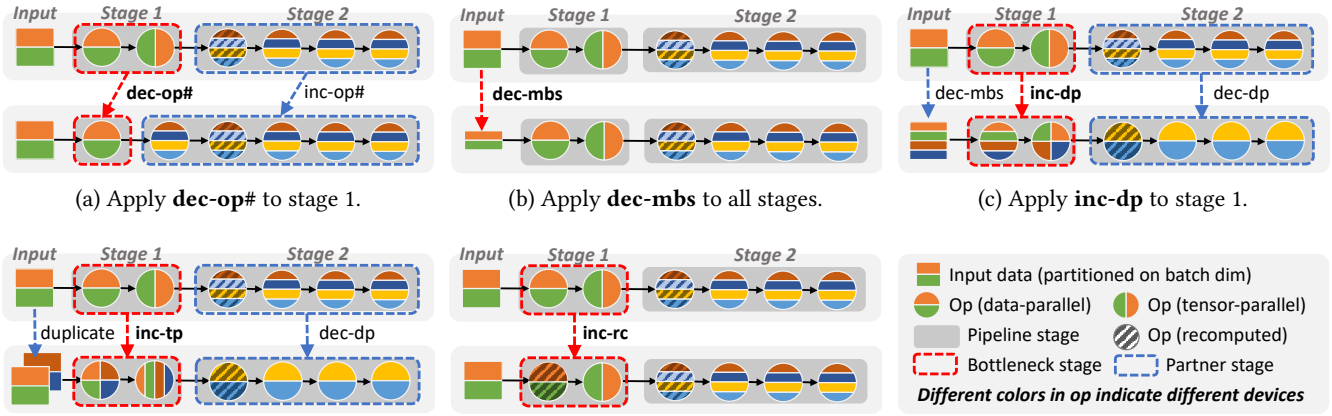
Heuristic-1 prioritizes memory over other resources to ensure "safety-first" and avoid inexecutable configurations.

3.2 How to alleviate the bottleneck for better configuration

Upon identifying a bottleneck, Aceso begins to look for a better configuration through bottleneck alleviation. Aceso leverages the idea of ***Resource Trading*** to alleviate resource bottlenecks. Bottlenecks are usually caused by a lack of certain resources, and so we can trade the surplus resources for the lacking ones (similar to the classic "Trading Storage for

ID	Primitive	Comp	Comm	Mem	Arguments	Value range	Mechanism	Description
1	inc-op#	↗	⇒	↗	stage-op#	[0, op#]	Pipeline parallelism	Increase/decrease number of operators in *a pipeline stage.*
2	dec-op#	↘	⇒	↘				
3	inc-mbs	↘	⇒	↗	microbatch size	[1, batchsize]	Pipeline parallelism	Increase/decrease microbatch size in *all the pipeline stages.*
4	dec-mbs	↗	⇒	↘				
5	inc-dp	↘	↗	↘	concurrency	[1, GPU#]	Data parallelism	Increase/decrease data-parallel size of *one or a group of operators.*
6	dec-dp	↗	↘	↗				
7	inc-tp	↘	↗	↘	concurrency	[1, GPU#]	Tensor parallelism	Increase/decrease tensor-parallel size of *one or a group of operators.*
8	dec-tp	↗	↘	↗				
9	inc-rc	↗	⇒	↘	recompute-op#	[0, op#]	Recomputation	Increase/decrease number of recomputed op in *a pipeline stage.*
10	dec-rc	↘	⇒	↗				

Table 1. Reconfiguration primitives table. (**Comp**, **Comm** and **Mem** show each primitive's impact on the consumption of three resources: computation, communication, and memory, with one of the trends: ↗ increase, ⇒ unchanged, ↘ decrease.)



(a) Apply **dec-op#** to stage 1. (b) Apply **dec-mbs** to all stages. (c) Apply **inc-dp** to stage 1.

(d) Apply **inc-tp** to stage 1. (e) Apply **inc-rc** to stage 1.

Figure 5. Illustration of reconfiguration primitives. An operator with two colors indicates it is split and assigned to two devices. Some primitives may implicitly involve device re-arrangement, which is shown by the colors.

Computation" idea). Resource trading can be achieved by reconfiguring some mechanisms of the configuration. However, it is challenging to analyze the resource impacts of complex reconfigurations and find a reconfiguration with performance improvement among so many possibilities. We simplify the problem by summarizing five pairs of basic reconfigurations, which are called ***Reconfiguration Primitives***. Aceso can utilize these reconfiguration primitives to alleviate the resource bottleneck directly.

We apply the resource trading idea during the search process when choosing among all available reconfiguration primitives. The resource impact is easy to analyze because each reconfiguration primitive only involves adjustments to one single mechanism. One-hop adjustment using primitives typically cannot immediately result in a better configuration than the initial one. Therefore, we further propose a ***Multi-hop Search*** algorithm to efficiently search a sequence of reconfiguration primitives until a better configuration is found.

3.2.1 Reconfiguration Primitives. We summarize all the reconfiguration primitives in our search space in Table 1 and further illustrate them in Figure 5. As shown in the table, different reconfiguration primitives have different impacts on resource utilization. It is difficult to find "free lunch" reconfigurations that reduce all resource consumption. Rather, most trade the consumption of one resource for another, meaning, given a configuration with resource utilization information of its bottleneck, we can query the table to efficiently find reconfiguration primitives to alleviate the bottleneck accordingly. All the listed primitives cover the common options available for bottleneck alleviation in our practice. Additionally, Aceso can be extended with new primitives for future research.

All reconfiguration primitives are semantic-preserving, which means they will not affect the convergence of training. For example, when the concurrency of data parallelism is increased for a certain stage, the microbatch size for each GPU will be decreased accordingly to maintain the aggregated microbatch size, as shown in Figure 5 (c).

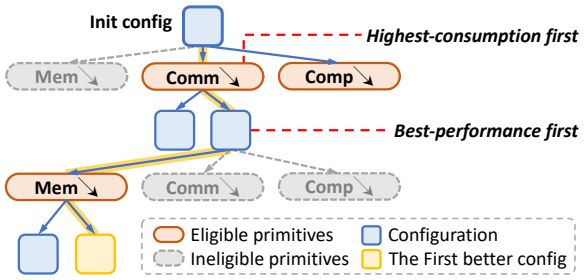


Figure 6. Example of primitive search.

Arguments. Each primitive is applied to modify one argument of a configuration. The argument value is chosen from a given value range, as shown in Table 1. For some primitives, the available values are quite limited, and we can enumerate them directly. For other primitives with more values in the range, e.g., the value range is [0, op#] for `inc/dec-op#` and `inc/dec-rc`, enumerating is inefficient. We leverage our performance model to quickly evaluate and choose the argument value with greedy-based approaches. Please refer to §4 for more details.

Granularity. In practice, the applied granularity of each primitive can be different. There are three types of granularities: op-level, stage-level, and model-level. For example, `inc/dec-dp` can be applied at both the op-level and stage-level, i.e., increasing the data-parallel concurrency of one operator or all the operators in the stage. Some other primitives can only be applied at the model-level, e.g., `inc/dec-mbs`, because in a pipelined system, changing the microbatch size of any stage will change the number of microbatches, thus affecting all the other stages. For the efficiency of the search, we apply all primitives at the stage-level or model-level during each search iteration and add a fine-tuning process after each iteration to find possible op-level re-configurations.

Partner primitives and partner stages. In Aceso, some primitives cannot be applied alone because they involve modification to at least two stages, requiring the application of a *partner primitive* to a corresponding *partner stage*. In Aceso's search space, such primitives (and their partner primitives) include: `inc-op#` (`dec-op#`), `inc-dp` (`dec-dp/tp`) and `inc-tp` (`dec-dp/tp`), as shown in Figure 5(a)(c)(d). When choosing the partner stage, Aceso chooses the one with the most available resources required by the bottleneck stage.

3.2.2 Which Primitive to Apply?

Once we have obtained the primitive table, we query the table to find eligible primitives with the idea of resource trading. In cases where there are multiple eligible primitives, we further prioritize the ones with higher potential in performance improvement.

Finding eligible primitives. Given a configuration with resource utilization information of its bottleneck, we query Table 1 to find reconfiguration primitives that can alleviate the bottleneck by reducing the usage of the constrained resource. This is efficient because only a small number of primitives satisfy the resource usage requirement, significantly saving effort to explore unqualified primitives.

Ranking eligible primitives. We may have several eligible primitives during each round of the search. We rank the exploration order of the primitives according to the primitives' potential in performance improvement: 1) When a bottleneck is caused by multiple resources, our priority is to first reduce the one with the highest *consumption proportion*, which is defined as the consumed amount in the present stage divided by the total consumed amount in all the stages. Primitives that can decrease the consumption of such resources have a higher potential to alleviate the bottleneck. This method resembles "gradient-descent", a well-known iterative optimization algorithm. 2) When there are multiple eligible primitives for alleviating the same resource bottleneck, we estimate the configuration performance after each primitive is applied separately, where better performance indicates higher potential. In Aceso, we summarize the above approach and use the following heuristic to find the primitive for reconfiguration:

Heuristic-2: *During the search, we explore primitives in the following order:*

- ***Highest-consumption first:*** *If multiple resources cause the bottleneck, we first alleviate the one with the highest consumption proportion.*
- ***Best-performance first:*** *If multiple primitives can reduce the consumption of the same resource, we first choose the one with the best estimated performance.*

Example. Figure 6 shows an example of primitive search in Aceso. For the initial configuration, we identified that its bottleneck is caused by computation or communication, and we explored communication-decreasing primitives first due to higher consumption proportion. From the reconfiguration table, we found 2 primitives that could decrease communication usage. Next, we assessed the performance when the two primitives are applied separately, and selected the one with better performance.

3.2.3 Multi-hop search.

Simply applying one reconfiguration primitive can alleviate a bottleneck, but usually causes a new bottleneck in another stage, and the new bottleneck stage may perform even worse than the previous one. To alleviate the new bottleneck, we may perform another round of search. After several rounds, we may ultimately discover a better configuration after applying a sequence of primitives to the original one. If not, we backtrack and try the other eligible primitives, as described in Heuristic-2. We call this procedure *Multi-hop Search*. For example, to alleviate memory pressure by increasing tensor-parallel concurrency (allocating more GPUs) in the bottleneck stage, the pipeline may become very imbalanced as the stage that contributes GPUs gets much slower, resulting in worse system performance

than before. But if we apply an additional `dec-op#` primitive, the pipeline can be re-balanced and may demonstrate better performance. In such a case, we make a 2-hop search to obtain a better configuration. As shown in Figure 6, if the "Best-performance first" configuration does not yield better results than "Init config", the multi-hop search continues its search for primitives based on the current configuration and finds a better configuration with a memory-decreasing primitive.

Depth of search. Reconfigurations with more hops of primitives have better improvements due to better expressiveness. But large hop length may hurt search efficiency and make each iteration more costly. We provide a hyper-parameter *MaxHops* to limit the hop length of a multi-hop primitive and we will explain how we set this hyper-parameter in §5.2.

Exploring secondary bottlenecks when failed. In some cases, we may fail to find any better configuration after searching all the configurations, given the limited depth. Then we will try to search starting from the secondary bottlenecks. For example, given that a memory-intensive operator dominates the top-1 bottleneck stage, and the adjacent stages are also under memory pressure, if only `"dec-op#"` works for the bottleneck stage, Aceso would fail because the adjacent stages do not have enough memory for `"inc-op#"`. But the adjacent stage may have other options to remove the memory pressure and then help alleviate the top-1 bottleneck.

3.3 Performance model

Aceso relies on resource utilization information to identify the bottleneck and choose reconfiguration primitives. This information includes computation time, communication time, and memory consumption. In addition, Aceso also needs to know the iteration time of configurations to evaluate the performance improvement. Aceso does not require a completely accurate performance model, as it only needs to compare configurations instead of relying on the absolute predicted values. We built a profiling-based performance model for Aceso, where we profiled basic information for each operator and composed them to predict resource utilization information of each stage and the iteration time of the full model. Aceso profiles the time and memory usage of each operator under different partition methods, as well as the collective communication time under different communication group sizes. The profiled database can be reused by the search for models that contain the same operators or the search for the same models with different search settings.

Memory consumption prediction. We estimate peak memory consumption as the sum of model parameters, activation size, and optimizer states. In pipeline parallelism with 1F1B scheduling [28], the earlier pipeline stages keep more copies of activation. Memory consumption of the ith pipeline stage is defined as: (p is the number of pipeline stages)

$$Memory_i = M_{param_i} + M_{act_i} * (p - i) + M_{opt_i} \quad (1)$$

Aceso also follows [20] to calculate peak memory usage when recomputation is enabled.

In addition to the ideal memory usage described above, the training framework may require extra memory. We observed that the major source of extra memory consumption is the PyTorch memory allocator, which takes charge of memory management. When an operator completes its computation, the memory used by that operator might not be immediately released. Instead, the allocator may retain it to expedite future memory allocations. Given the intricacy of the memory allocator and the risk of underestimating memory consumption, which can result in OOM configurations, we opt to overestimate the amount of reserved memory. Specifically, we consider the maximum amount of memory used among all operators in a pipeline stage as the estimated extra memory usage for that stage.

Iteration time prediction. Aceso employs pipeline parallelism (1F1B scheduling as [28, 51]) combined with data and tensor parallelisms. In a pipeline training system, overall performance is bottlenecked by the stage with the longest execution time. Therefore, we predict the time required for each stage and take the longest as the iteration time for the entire model. We calculate the time for each stage by considering the time for three phases in the pipeline, i.e., warmup, cooldown, and steady state:

$$T_{stage_i} = T_{warmup_i} + T_{steady_i} + T_{cooldown_i} \quad (2)$$

Pipeline warmup time equals to the forward computation time of a microbatch through all stages, while cool-down time equals to the corresponding backward computation time. In the steady states of the pipeline, N microbatches are computed one by one in both forward and backward passes, with communication occurring between consecutive pipeline stages. When recomputation is enabled, the operator's backward time will increase by its corresponding forward computation time, which is not included in the equation for simplicity.

4 Implementation

We prototyped Aceso with 10K lines of Python code on top of Megatron-LM (v2.4) [43], incorporating additional support for Aceso's search space. We ensured the correctness of our implementation by comparing the output with that of the original Megatron-LM.

In the following subsections, we describe more implementation details of the search algorithm of Aceso, including choosing primitive argument values, the fine-tuning process, and various optimizations.

4.1 Choosing primitive argument values

There may be many choices for the argument value when applying certain primitives, including `inc-op#`, `dec-op#`, `inc-rc`, and `dec-rc`. Since the four primitives modify the number of operators/recomputed operators in a pipeline stage, this choice not only involves *how many* but also *which* operators to move out or recompute. We leverage our performance model and greedy-based approach to make the choice efficiently.

Inc/dec-op#. In terms of *which* operators to move, due to the data dependency of DNN models, we can only move a sequence of consecutive operators at the beginning of a pipeline stage to an earlier stage, or a sequence of consecutive operators at the end to the latter stage. As for the direction of movement (earlier stage/latter stage), we opt for the one that is closer to the idlest stage. In terms of *how many* operators to move, we first try to move out as few as possible operators to alleviate the bottleneck under a tight goal, e.g., decrease the execution time of the bottleneck stage to the time of the adjacent stage. If the goal cannot be achieved after evaluating all the possibilities with our performance model, we set a looser goal until it can be achieved.

Inc/dec-rc. With regards to *how many* and *which* operators to recompute when the bottleneck stage exceeds the device memory limit, we employ a greedy approach by choosing the operators with the largest activation size for recomputation and carry out the process until memory consumption is under the device limit.

4.2 Fine-tuning process.

After each search iteration, we support optional op-level adjustments to fine-tune some parallelism arguments for each operator, including:

Flexible combination of tp/dp concurrency inside a pipeline stage. Altering the tp/dp concurrency within a stage can potentially enhance performance, but it may also introduce additional communication overhead. For instance, in a stage where the first half involves 2-way dp + 2-way tp and the second half involves 4-way tp, an all-gather operation might be necessary between the two halves. Therefore we prefer to minimize the frequency of tp/dp concurrency changes within a stage. We assess the impact of increasing tp/dp starting from each operator until the end of the stage and keep the change if it leads to a better configuration.

Flexible tensor-parallel dimension of each operator. Aceso does not explore all possible partition dimensions for each operator and instead chooses the tensor parallelism dimension given several partition options. For example, in Transformer-based models, we partition MatMul using two options (row-wise and column-wise) and follow Megatron-LM's solution to set the initial partition dimension. For the Wide-Resnet model, we partition convolution operators using two options (input-channel and output-channel) and set out-channel as the initial partition dimension. During the fine-tuning process, we check the partition options for each operator to determine if there is a better dimension for tensor parallelism.

4.3 Search algorithm optimizations

Combination of multiple reconfiguration primitives. We observed that certain reconfiguration primitives usually work together. Therefore, we combined them in our implementation so that we could use a smaller search depth to find more complex reconfigurations. These combinations include:

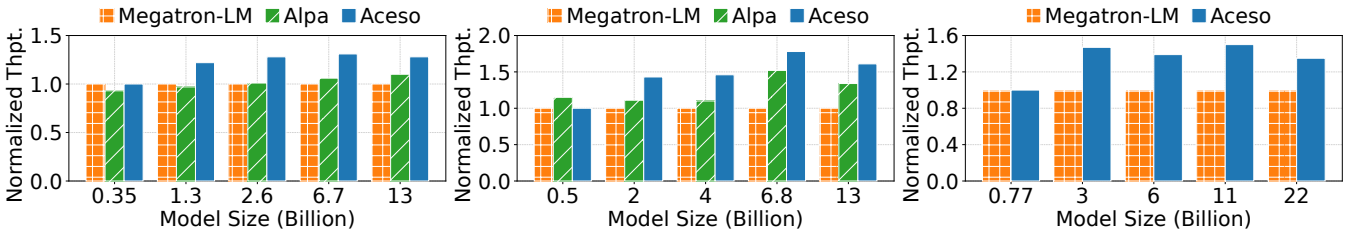
- Attaching `inc/dec-rc` to all other primitives. We do this because it's usually necessary to check if the recomputation configuration still works if the memory consumption changes.
- Combining multiple `inc/dec-op#` primitives. We observed that simply applying one `inc/dec-op#` primitive is usually inefficient because it can only migrate operators between adjacent pipeline stages but the bottleneck stage and the idlest stage may at a distance, so we do it in a relay manner: if we identify stage i as the bottleneck and stage $i+2$ as the idlest one, we would first migrate operators from stage i to stage $i+1$, then from stage $i+1$ to stage $i+2$.

Parallel search of configuration under different pipeline stage numbers. We do not include "inc/dec pipeline stage#" as one of our reconfiguration primitives due to complexity. So, we need to search for the best configuration under different pipeline stage numbers. There are no dependencies among searches with different stage numbers, which makes them suitable for parallelization. With parallelization, the search overhead of Aceso is essentially equivalent to searching for one pipeline stage number.

Deduplication. There is a chance that the search would run into configurations that have been visited before. Aceso leverages a configuration-semantic aware hashing method to index configurations and avoid exploring visited ones.

5 Evaluation

We evaluated Aceso on its performance in training large-scale DNN models and the cost of running a configuration search. Furthermore, we investigated the efficiency of Aceso's search algorithm through a series of ablation studies. Finally, we presented two case studies to demonstrate the configurations discovered by Aceso's search algorithm. Our evaluation was performed on a cluster with 32 NVIDIA V100 (32GB) GPUs. Each server was equipped with 8 GPUs. GPUs are intra-server connected through NVLink and inter-server connected through Infiniband (100Gb/s per server) network [31].



(a) GPT-3 Model. (b) Wide-Resnet Model. (c) T5 Model.

Figure 7. (Exp#1) The training throughput of GPT-3, Wide-Resnet and T5 models.

Table 2. Model specifications.

Model	Parameter# (billion)	Precision	Batch size	Seq length
GPT-3	0.35, 1.3, 2.6, 6.7, 13	FP16	1024	2048
T5	0.77, 3, 6, 11, 22	FP16	1024	2048/512
Wide-Resnet	0.5, 2, 4, 6.8, 13	FP32	1536	224×224×3

Benchmark models. To conduct the evaluation, we selected 3 representative large-scale deep learning models from different domains, and with different model structures.

- **GPT-3** [3] is a natural language model that stacks multiple homogeneous transformer decoder layers.
- **T5** [40] is a natural language model with a heterogeneous and imbalanced structure that contains both transformer encoder and decoder layers.
- **Wide-Resnet** [50] is a convolutional vision model that is derived from ResNet [13] but with wider convolution layers.

Each model has a set of different parameter sizes. Table 2 presents the basic information of each model with details in [3, 40, 50]. For T5 model's sequence length, 2048 and 512 are for encoders and decoders respectively. For Wide-Resnet, we show the input image size in the sequence length column.

Baseline systems. We compared Aceso with two state-of-the-art distributed deep learning training systems, Megatron-LM (v2.4) and Alpa (v0.1.5), to evaluate its training performance and automated configuration searching cost.

- **Megatron-LM** combines tensor parallelism, data parallelism, pipeline parallelism, and recomputation. Accordingly, there are five configuration options: *tp*, *dp*, *pp*, *b*, *recomp*. *tp*: tensor parallelism concurrency, *dp*: data parallelism concurrency, *pp*: pipeline parallelism stage number, *b*: microbatch size, *recomp*: recomputation enabled or not. Megatron-LM sets these options globally, e.g., all different model layers share the same data parallelism concurrency. Megatron-LM does not support automated configuration search. To maximize its performance as a strong baseline, we performed a grid search over all these options using Aceso's performance model to find the best configuration for Megatron-LM.
- **Alpa** supports automated parallelism over all the mechanisms in Megatron-LM but allows the configuration of different operators flexibly instead of globally. Alpa first groups all the operators into l layers, then finds the configuration using its solver. Configurations such as microbatches, and model-wise recomputation are set manually in Alpa, so we performed a grid search for the parameters: l, b, *recomp* as a complete automated configuring.

5.1 Training Performance

We evaluated the training performance of Aceso, Megatron-LM and Alpa.

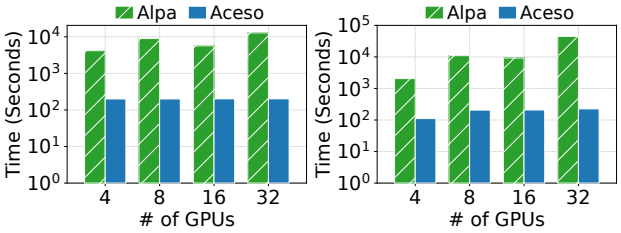
For Aceso, we set the same search time (200s) for all the experiments for simplicity. Likewise, we found the value of *MaxHops* insensitive to changing experiments, so we used the same value (7) across all the experiments empirically. We initialized the search with an evenly partitioned configuration. To maintain load balanced in data parallelism and tensor parallelism and ensure memory alignment for better performance [32], we limited the concurrency of data parallelism and tensor parallelism as powers of two. To mitigate the prediction error of our performance model, we evaluated the top 5 configurations returned by the Aceso search algorithm and kept the one with the best performance.

For Megatron-LM, we obtained the performance results for Wide-Resnet and T5 using our runtime, the modified Megatron-LM. This is because Wide-Resnet is not supported in Megatron-LM, and pipeline parallelism for T5 models is supported in the newer version. To draw a fair comparison, we chose to simulate Megatron-LM's plan for T5 models in the version we worked on.

For Alpa, we turned on kernel tuning for Wide-Resnet, since the default setting (disable) hurts throughput significantly.

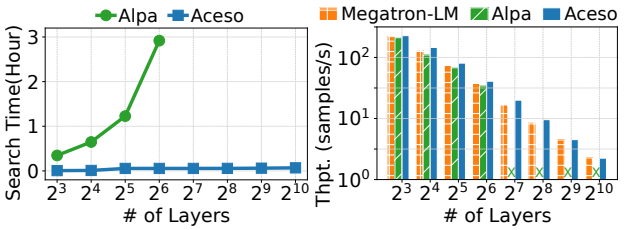
For the 1-GPU setting, we ran all the systems under the same configuration found by Alpa.

(Exp#1) Throughput comparison. Figure 7 compares system throughput (normalized) on three models with different settings. We increased GPU number with model size increasing and used 1, 4, 8, 16 and 32 GPUs for the five selected model sizes, respectively. Aceso found better configurations among all the settings. The throughput of single GPU settings shows that different systems have similar execution efficiency. For GPT-3, the well-studied homogeneous



(a) GPT-3 Model. (b) Wide-Resnet Model.

Figure 8. (Exp#2) Search cost comparison.



(a) Search Cost. (b) Training Throughput.

Figure 9. (Exp#3) Search cost and training throughput of different systems. (×: execution fails.)

model, Aceso achieved up to 1.27× speedup over the state-of-the-art system Alpa. For Wide-Resnet, Aceso achieved up to 1.33× speedup over Alpa, and up to 1.78× speedup over Megatron-LM. For the T5 model, we focused on comparison with Megatron-LM, as there is no official implementation in Alpa. Aceso achieved up to 1.50× speedup. Considering that an end-to-end training usually involves hundreds of thousands of iterations and weeks of time [1], such throughput improvements can significantly cut down training time by several days or even more. We have also provided TFLOPS in the Appendix as a reference for absolute performance.

Performance improvement analysis. Aceso finds better configurations because it searches in a larger configuration space. Compared with Alpa, Aceso additionally supports 1) op-level on-demand recomputation, and 2) a more flexible partition plan inside a pipeline stage. For the second point, Alpa also supports in-stage flexible partition plans for different operators, but they use a simple cost estimator in the ILP solver when searching for the best partition plan within a single pipeline stage. In their estimator, the computation time of all operators is treated as 0, the difference in computation time among different partition plans is ignored, and only communication time is considered. Such simplification may overlook certain configurations where flexible partition inside one pipeline stage results in performance gains in computation time. For Megatron-LM, Aceso also benefits from the same two aspects, with the additional benefit of supporting 3) fine-grained op-level partition of pipeline stages. We will show some examples of configurations found by Aceso in §5.4.

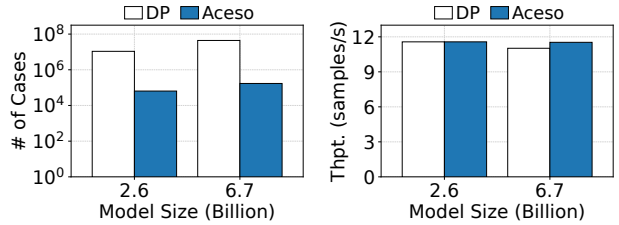
(Exp#2) Search cost comparison. We only compare Aceso with Alpa on search cost since the original Megatron-LM lacks automated search capabilities. Both Aceso and Alpa need to profile a performance database in advance. This preprocessing step is not included in the search cost because it is only done once and shared across experiments. More details are provided in Section 5.3. However, Alpa uses on-demand profiling, which is required in each experiment and thus is included in the search cost. Figure 8 shows the configuration search cost of Aceso and Alpa. Among all the cases, Aceso uses less than 5% of the time used by Alpa. Aceso can find good solutions in such a short time because it only explores a small part of the whole configuration space and utilizes an efficient performance model instead of costly real deployment. Alpa breaks the whole search space into two sub-spaces and groups operators to reduce search time. However, it still requires a considerable amount of time because Alpa needs to repeatedly compile and profile a lot of different XLA kernels [11], for which building an accurate performance model is more difficult. Alpa's search cost is not directly related to model size, but the hyper-parameter l – the number of layer groups. This explains why Alpa's cost goes down at the 16-GPU cases — because the predefined l is smaller in those cases. Although Alpa has reduced search space, the complexity of mathematical solutions still increases with the scale of the problem, which makes it hard to scale to larger models. We will discuss the scale performance of Alpa and Aceso next.

(Exp#3) Scalability to 1K layers. Current DNN models have scaled up to 1,000 layers [23, 47]. Mathematical programming solutions suffer from configuration search costs when the model layer increases. We evaluated the search cost of Aceso and Alpa with up to 1K-layer transformer models with hyper-parameters settings from [47] over 8 GPUs. As shown in Figure 9, Aceso can always find solutions while Alpa failed compilation when the layer number grows larger than 64. For layer numbers under 64, Alpa's search overhead increases linearly with increasing layer numbers, while Aceso always finishes the search under the given budget (200 seconds). For all model sizes Alpa can support, Aceso achieves 1.2× average speedup on training throughput.

5.2 Search Algorithm Ablation Study

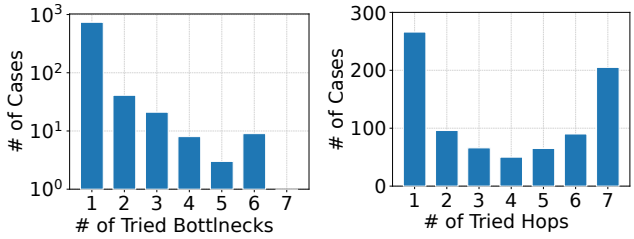
In this section, we study why Aceso outperforms the baseline systems and how its search settings impact performance.

(Exp#4) Exploration efficiency. To achieve the same performance, Aceso's search algorithm only needs to explore much fewer configurations. To compare Aceso's search approach to a mathematical programming approach, we implemented a dynamic programming (DP) solution and compared the number of explored configurations and the best configuration performance. Considering the exponentially large space, we carried out some pruning in the DP solution, such



(a) Explored Cases. (b) Training Throughput.

Figure 10. (Exp#4) Search overhead and performance of found cases in dynamic programming and Aceso.



(a) Tried Bottlenecks Distribution. (b) Hop Number Distribution.

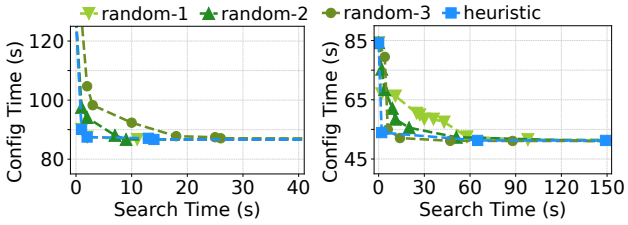
Figure 11. (Exp#5) Distribution of the number of tried bottlenecks and number of hops in one search iteration before achieving effective improvement.

as limiting the maximum number of operators at each stage, the maximum microbatch size, and the maximum tp/dp size. We used the same performance model in both approaches for a fair comparison.

As shown in Figure 10(a), the DP solution explores 10^7 and $4.3 * 10^7$ configurations for GPT-3 2.6B and 6.7B, respectively, while Aceso only explores 1% of them. We executed the configurations found by DP and Aceso in our runtime. As shown in Figure 10(b), for GPT-3 2.6B model, both approaches found the same configuration, demonstrating the same performance. For GPT-3 6.7B, DP and Aceso found configurations with similar estimated performance but Aceso achieved slightly better performance in runtime. This is because our performance model is not completely accurate, the best configuration predicted by the performance model may not be the best in real execution. We will evaluate the prediction accuracy later in this section.

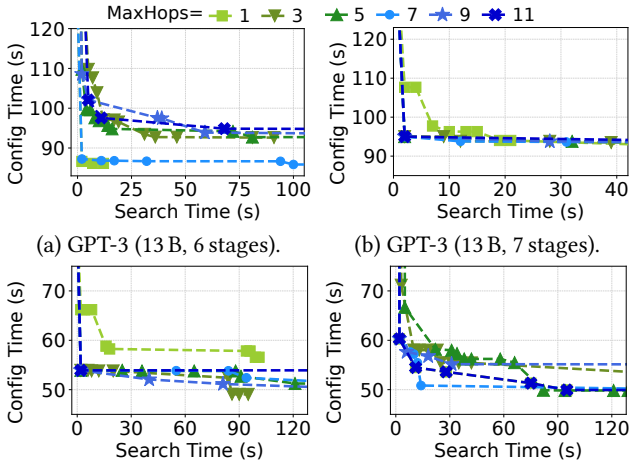
(Exp#5) Heuristic efficiency. Aceso's search algorithm uses heuristic-1 to identify bottlenecks and heuristic-2 to choose reconfiguration primitives. We evaluated the efficiency of the two heuristics separately.

For heuristic-1, if Aceso fails to find any better configurations starting from the identified bottleneck within the hop length, it would start another multi-hop search from the next bottleneck. The number of bottlenecks it tries on before finding improvements shows the efficiency of heuristic-1. We summarized the number of bottlenecks tried in each search iteration for the experiments in §5.1. As shown in Figure 11



(a) GPT-3 (13 B, 8 stages). (b) Wide-ResNet (13 B, 8 stages).

Figure 12. (Exp#5) Convergence trends w/&w/o heuristic-2.



(a) GPT-3 (13 B, 6 stages). (b) GPT-3 (13 B, 7 stages).

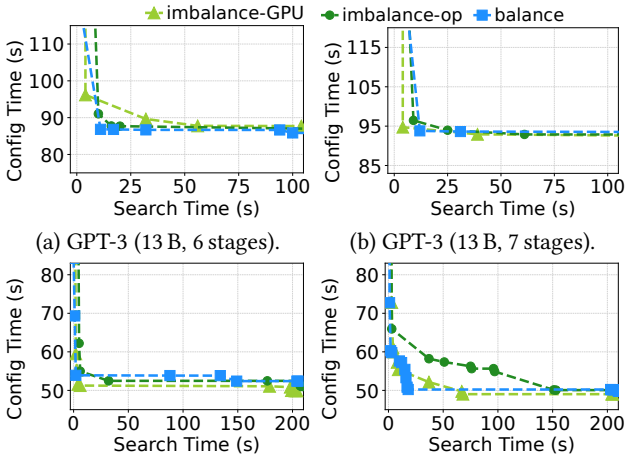
(c) Wide-ResNet (13 B, 8 stages). (d) Wide-ResNet (13 B, 9 stages).

Figure 13. (Exp#6) Convergence trends under different maximum hop length.

(a), heuristic-1 can effectively identify bottlenecks. In 90% of all the 819 search iterations, it finds the right bottleneck at the first attempt to improve performance.

For heuristic-2, we demonstrated its efficiency by comparing the search convergence trends with/without heuristic-2. The convergence trend shows how the estimated execution time of the best-found configuration changes over search time, as shown in Figure 12. When heuristic-2 is not enabled, we would randomly choose primitives to explore during the search, and we ran random search 3 times. Searching with and without heuristic-2 can both find similar configurations given enough search time budget, but the random search may lead to an inefficient search path and result in a sub-optimal case if the search time budget is very limited. We also show the number of attempted hops before finding improvements in Figure 11 (b), where 68% of search iterations require more than one hop to discover better configurations.

(Exp#6) Search efficiency under different hop lengths. As mentioned in § 3.2.3, a larger hop number brings better performance but hurts search efficiency when it gets too large. We show the search convergence trends under different maximum hop lengths (*MaxHops*) in Figure 13.



(a) GPT-3 (13 B, 6 stages). (b) GPT-3 (13 B, 7 stages).

(c) Wide-Resnet (13 B, 8 stages). (d) Wide-Resnet (13 B, 9 stages).

Figure 14. (Exp#7) Convergence trends with different initial configurations.

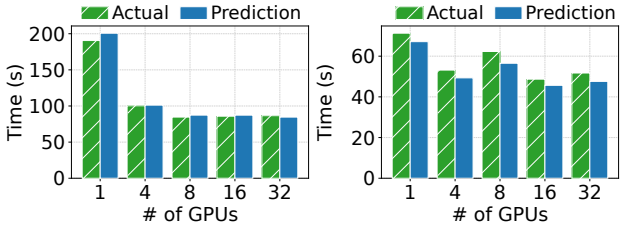
While smaller *MaxHops* may lead to a sub-optimal case (*MaxHops* = 1 in Figure 13 (c)), a larger *MaxHops* may spend too much time in some iterations, causing it to run out of time for further searches (*MaxHops* = 11 in Figure 13 (a)). The most suitable *MaxHops* for a given model and search time budget is difficult to determine in advance, so for our evaluation, we opted for a moderate size (*MaxHops* = 7) in our evaluation.

(Exp#7) Robustness over initial configuration. Aceso's search algorithm starts from a default configuration with a balanced partition and minimum microbatch size. To confirm the robustness of the initial configuration, we introduced two additional initial configurations: one with imbalanced workloads (imbalance-op) and another with imbalanced hardware allocation (imbalance-GPU). Figure 14 shows that Aceso's search algorithm can converge to a similar configuration despite having different initial configurations.

5.3 Performance Model Evaluation

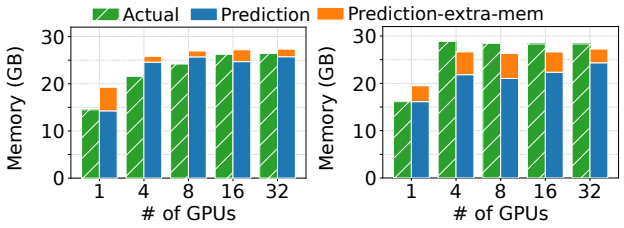
(Exp#8) Accuracy of iteration time prediction. We compared predicted execution time with actual execution time to evaluate the accuracy of Aceso's performance simulator. As shown in Figure 15, Aceso achieves strong prediction accuracy with only an average error of 2.70% and 7.29% for the GPT-3 and Wide-Resnet models, respectively.

(Exp#9) Accuracy of memory prediction. We compared the predicted and actual memory consumption in Figure 16. As mentioned in §3.3, we always over-estimate the extra memory consumption to avoid out-of-memory configurations. This may result in over-estimation in a few cases, e.g., the 1-GPU cases. The average prediction errors for GPT-3 and Wide-Resnet are 14.26% and 9.14% respectively (and are as small as 9.57% and 6.23% without 1-GPU cases).



(a) GPT-3 Model. (b) Wide-ResNet Model.

Figure 15. (Exp#8) Comparison of predicted and actual iteration time.



(a) GPT-3 Model. (b) Wide-ResNet Model.

Figure 16. (Exp#9) Comparison of predicted and actual memory consumption.

Profiling overhead. As mentioned in §3.3, we profiled the database for a group of models only once. For each group of models (GPT-3, T5, and Wide-Resnet), we ran all the different operators 50 times and calculated the average execution time. We used similar methods to profile the collective communication time. The total profiling time (using 4 nodes) is 11 minutes, 5 minutes, and 1.5 hours for GPT-3, T5, and Wide-Resnet respectively. Wide-Resnet requires more time due to having a larger number of different operators in the model. We found that the profiling is sufficiently accurate from the evaluation. Spending longer profiling time only gives marginal improvements on the accuracy of the model. Currently, we are running the profiling of operators sequentially. The profiling overhead can be highly improved with good parallelization. We leave this as future work.

5.4 Case Study

Aceso configuration for GPT-3: uneven pipeline stages. GPT-3 models are partitioned into pipeline stages evenly by Megatron-LM and Alpa, consisting of homogeneous layers. In addition, Alpa and Megatron-LM prioritize data parallelism to save communication cost [29, 51]. For example, for the GPT-3 1.3B model on 4-GPU, the best configuration found by Alpa and Megatron-LM was 4-way data parallelism with recomputation enabled for all the operators. In contrast, Aceso found the configuration with 4-way pipeline parallelism and only a few operators recomputed in the first pipeline stage. Less recomputation is needed because

pipeline parallelism reduces the memory pressure. In addition, Aceso partitioned the model into uneven pipeline stages, where the first and last stages contained fewer operators than the middle two stages. The uneven partition leads to a balanced pipeline due to the recomputation cost in the first stage and the loss computation cost in the last stage. This type of configuration falls outside of the configuration space of Megatron-LM or Alpa.

Aceso configuration for Wide-Resnet: different operators adopt diverse parallelism settings. Both Alpa and Aceso partitioned Wide-Resnet 6.8B model over 16-GPU into 3 pipeline stages (4,4,8 GPUs). In the last stage, Alpa adopted 8-way tensor parallelism among all operators, whereas Aceso selectively combined 2-way data parallelism with 4-way tensor parallelism for specific operators. This decision stemmed from the fact that only the last three blocks needed 8-way tensor parallelism because of substantial memory consumption. Applying 8-way tensor parallelism to other operators would harm their computational efficiency.

6 Related Work

Parallel deep learning training mechanisms and systems. Parallelisms are used for distributed DNN training systems. TorchDDP [9] and Horovod [46] support data parallelism. ByteScheduler [36] and DeepSpeed [42] extend data parallelism with communication and memory optimizations. Tofu [48], Flexflow [18], and GSPMD [49] leverage tensor parallelism to distribute model weight across devices. Dapple [10] and Megatron-LM [43] also leverage pipeline parallelism to partition models across devices. Besides, recomputation [5, 20] is proposed to reduce memory consumption and improve training performance, which is adopted in both Megatron-LM and Alpa [51]. All these mechanisms can be modeled as resource trade-offs in Aceso for joint considerations.

Automatic search for distributed DNN execution. The pipeline parallelism methods [44, 45, 51] usually leverage dynamic programming to decide operator placement for balanced pipeline stages. ML-based solutions such as reinforcement learning [27] are also considered for the same goal. FlexFlow [18] mainly considers the exploration of tensor parallelism strategy, without considering the combination of other mechanisms. It proposes an MCMC-based heuristic search that is tailored to minimize the communication among different operator partitioning strategies. To combine pipeline parallelism with tensor parallelism, Piper [44] predecides a well-designed tensor parallelism strategy and leverages dynamic programming for balancing pipeline stages. Alpa [51] further employs ILP solver and dynamic programming for searching intra-op parallelism and inter-op parallelism, respectively. For other techniques such as recomputation, DTR [20] adopts a greedy online algorithm to make the recompute choices for different operators. FTPipe [8] supports combined data and pipeline parallelisms targeting model fine-tuning instead of training on commodity hardware. Its automated configuring is achieved by selecting from candidate partitioning methods based on performance simulation. Aceso differs from these systems by tackling the problem from the view of bottleneck alleviation, so that it can jointly consider the various mechanisms. [27] uses the reinforcement learning (RL) approach to automate device placement of DNN. Besides device placement, Aceso needs to consider more variables in parallel DNN configuration problems, making it hard to solve with the RL approach.

Bottleneck analysis and resolving. Many works [4, 19, 33, 35, 37] in stream processing have leveraged resource-aware bottleneck analysis to improve system performance. PETS [37] proposes a tuning system to adjust Spark parameters by modeling resources as scores. R-Storm [35] considers the resource demand of different tasks and proposes a resource-aware scheduling algorithm to allocate different hardware resources for different tasks. Joker [19] resolves CPU utilization bottlenecks by adjusting resources allocated for replicating operators and pipeline stages on-the-fly. Different from Aceso, which focuses on GPU execution, they don't consider characteristics of DNN mechanisms for deep learning. For data-parallelism-only scenarios, dPro [14] is a toolkit that collects runtime traces to identify performance bottlenecks and explores optimization strategies. Compared with dPro, Aceso supports parallel training mechanisms more than data parallelism.

DNN performance prediction. Daydream [52] builds a kernel-level performance estimator to evaluate diverse DNN optimizations. In contrast, Aceso's performance estimator provides resource utilization for its configuration searching.

7 Conclusion

Emerging parallel training mechanisms are critical for large-scale DNN training but introduce a huge configuration space. Rather than conducting costly exhaustive searches over a performance-compromised sub-space, Aceso takes a different approach by conducting iterative bottleneck-alleviation searching. Along with analysis on unified re-configurations, Aceso significantly reduces search complexity while achieving better efficiency and scalability. With its low searching cost and high-quality results, we hope Aceso can contribute to the scaling of model training and adapt to dynamic resource scenarios.

Acknowledgments

We sincerely thank our shepherd Jongsoo Park and all the anonymous reviewers for their valuable feedback. We also thank Madan Musuvathi, Jun Huang and Haifeng Sun for their suggestions and help. Authors from Chinese Academy of Sciences were supported by the National Natural Science Foundation of China under Grant No.62172388.

References

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

[2] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. 2018. Benchmark analysis of representative deep neural network architectures. *IEEE Access* 6 (2018), 64270–64277.

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

[4] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 725–736.

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

[6] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. 2012. Large scale distributed deep networks. *Advances in Neural Information Processing Systems* 25 (2012).

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[8] Saar Eliad, Ido Hakimi, Alon De Jagger, Mark Silberstein, and Assaf Schuster. 2021. Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 381–396.

[9] Facebook. 2022. *PyTorch*. Retrieved May, 2022 from https://pytorch.org/

[10] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 2021. 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*. 431–445.

[11] Google. 2022. *XLA: Optimizing Compiler for Machine Learning*. Retrieved May, 2022 from https://www.tensorflow.org/xla

[12] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-efficient backpropagation through time. *Advances in Neural Information Processing Systems* 29 (2016).

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[14] Hanpeng Hu, Chenyu Jiang, Yuchen Zhong, Yanghua Peng, Chuan Wu, Yibo Zhu, Haibin Lin, and Chuanxiong Guo. 2022. dPRO: A Generic Performance Diagnosis and Optimization Toolkit for Expediting Distributed DNN Training. *Proceedings of Machine Learning and Systems* 4 (2022), 623–637.

[15] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*. 103–112.

[16] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems* 2 (2020), 497–511.

[17] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, et al. 2022. Whale: Efficient Giant Model Training over Heterogeneous GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 673–688.

[18] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. *Proceedings of Machine Learning and Systems* 1 (2019), 1–13.

[19] Basri Kahveci and Buğra Gedik. 2020. Joker: Elastic stream processing with organic adaptation. *J. Parallel and Distrib. Comput.* 137 (2020), 205–223.

[20] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2020. Dynamic Tensor Rematerialization. In *International Conference on Learning Representations*.

[21] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Reducing Activation Recomputation in Large Transformer Models. *arXiv preprint arXiv:2205.05198* (2022).

[22] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. 2019. Efficient rematerialization for deep networks. *Advances in Neural Information Processing Systems* 32 (2019).

[23] Guohao Li, Matthias Müller, Bernard Ghanem, and Vladlen Koltun. 2021. Training graph neural networks with 1000 layers. In *International Conference on Machine Learning*. PMLR, 6437–6449.

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

[25] Ze Liu, Han Hu, Yutong Lin, Zhuliang Yao, Zhenda Xie, Yixuan Wei, Jia Ning, Yue Cao, Zheng Zhang, Li Dong, et al. 2021. Swin Transformer V2: Scaling Up Capacity and Resolution. *arXiv preprint arXiv:2111.09883* (2021).

[26] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 10012–10022.

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

[28] 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. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

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

[30] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. 2009. Analytical modeling of pipeline parallelism. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 281–290.

[31] NVIDIA. 2020. *DGX-1*. Retrieved Aug. 2022 from https://www.nvidia.com/en-gb/data-center/dgx-systems/dgx-1/

[32] NVIDIA. 2022. *CUDA C Programming Guide*. Retrieved Aug. 2022 from https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[33] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 293–307.

[34] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2021. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350* (2021).

[35] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. 2015. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*. 149–161.

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

[37] Tiago BG Perez, Wei Chen, Raymond Ji, Liu Liu, and Xiaobo Zhou. 2018. Pets: Bottleneck-aware spark tuning with parameter ensembles. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–9.

[38] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. *arXiv preprint arXiv:1704.01444, 2017* (2018).

[39] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[40] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.* 21, 140 (2020), 1–67.

[41] Rajat Raina, Anand Madhavan, and Andrew Y Ng. 2009. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*. 873–880.

[42] 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.

[43] 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. *arXiv preprint arXiv:1909.08053* (2019).

[44] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. 2021. Piper: Multidimensional planner for DNN parallelization. *Advances in Neural Information Processing Systems* 34 (2021), 24829–24840.

[45] Jakub M Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. 2020. Efficient algorithms for device placement of DNN graph operators. *Advances in Neural Information Processing Systems* 33 (2020), 15451–15463.

[46] Uber. 2022. *Horovod*. Retrieved Aug. 2022 from https://horovod.ai/

[47] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Dongdong Zhang, and Furu Wei. 2022. Deepnet: Scaling transformers to 1,000 layers. *arXiv preprint arXiv:2203.00555* (2022).

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

[49] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. 2021. GSPMD: General and Scalable Parallelization for ML Computation Graphs. *arXiv preprint arXiv:2105.04663* (2021).

[50] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide residual networks. *arXiv preprint arXiv:1605.07146* (2016).

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

[52] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. 2020. Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 337–352.

A Absolute Performance

Our evaluation is performed in a cluster of 4 nodes, with each equipped 8 NVIDIA V100 (32GB) GPUs. We show TFLOPS per GPU of the Exp#1 in Table 3, 4, 5 as a reference of absolute performance. We got the FLOP count of GPT-3 and Wide-Resnet using Alpa's FLOP counter and calculated the FLOP count of T5 following Megatron-LM's methodology. Here we calculate effective TFLOPS, which does not include recomputation.

Table 3. GPT-3 TFLOPS per GPU

	0.35B	1.3B	2.6B	6.7B	13B
Megatron-LM	30.05	38.88	43.42	50.31	48.46
Alpa	27.79	37.64	43.81	53.34	53.34
Aceso	30.05	47.50	55.47	65.88	62.15

Table 4. Wide-Resnet TFLOPS per GPU

	0.5B	2B	4B	6.8B	13B
Megatron-LM	10.11	9.49	8.09	6.94	7.35
Alpa	11.60	10.58	8.86	10.51	9.88
Aceso	10.11	13.54	11.78	12.37	11.86

Table 5. T5 TFLOPS per GPU

	0.77B	3B	6B	11B	22B
Megatron-LM	31.66	32.56	31.37	27.85	27.60
Aceso	31.66	47.82	43.45	41.81	37.09

B Aceso Search Algorithm

We show the pseudo-code of Aceso search with Algorithm 1 and Algorithm 2. Algorithm 1 shows the high-level workflow of the iterative search, while Algorithm 2 shows the detail of the multi-hop search, in which *GetBottleneck*() and *GetPrimitives*() are implemented with heuristic-1 and heuristic-2.

Algorithm 1: AcesoSearch

```
Input: init_config: initial configuration, TimeBudget: time
       budget, MaxHops: max search depth
Output: best_config: the searched best configuration
unexplored_configs ← {init_config}
config ← init_config
best_config ← init_config
init_time ← get_time( )
while get_time( ) - init_time < TimeBudget do
    hop_index ← 0
    init_perf ← Estimate(config)
    new_config ← MultiHopSearch(config, hop_index,
      MaxHops, unexplored_configs, init_perf)
    if new_config is not None then
        config ← new_config
        best_config ← new_config
    else
        config ← GetBestOf(unexplored_configs)
return best_config
```

Algorithm 2: MultiHopSearch

```
Input: config: input configuration, hop_index: current hop
       index, MaxHops: max hop length,
       unexplored_configs: unexplored configs, init_perf:
       performance of initial config
Output: config: The searched best configuration, return
        None when fail within given hop length.
unexplored_configs.remove(config)
if hop_index == MaxHops then
    return None
else
    bottleneck ← GetBottleneck(config)
    primitive_groups ← GetPrimitives(config, bottleneck)
    for primitives ∈ primitive_groups do
        new_config_list ← []
        for prim ∈ primitive_groups do
            new_config ← Apply(config, bottleneck, prim)
            perf ← Estimate(new_config)
            unexplored_configs.put(new_config)
            new_config_list.append(new_config)
            if perf < init_perf then
                return new_config
        new_config_list.sort()
        for new_config ∈ new_config_list do
            new_config_more_hops ←
              MultiHopSearch(new_config, hop_index++,
              MaxHops, unexplored_configs, init_perf)
            if new_config_more_hops is not None then
                return new_config_more_hops
return None
```

C Artifact Appendix

C.1 Abstract

Aceso is a scalable parallel-mechanism auto-configuring system for large-scale DNN models. In this artifact, we present: (1) how to evaluate functionality with a small set of GPUs; (2) how to reproduce the results reported in the paper.

C.2 Description & Requirements

C.2.1 How to access. Github repo: https://github.com/microsoft/SuperScaler/tree/EuroSys24AE. DOI: 10.5281/zenodo.10077042.

C.2.2 Hardware dependencies. The hardware we used in the evaluation is a cluster of 4 nodes, each node is equipped with 8 NVIDIA V100(32GB) GPUs. GPUs are connected through NVLink (intra-server) and Infiniband (100Gb/s) network (inter-server).

It requires the same cluster to fully reproduce all the evaluation in the paper. However, a small set of GPUs, e.g., 4 V100 GPUs, are already enough to check Aceso's functionality.

C.2.3 Software dependencies. Aceso requires CUDA 11.6, Python 3.7, PyTorch 1.12, and apex 0.1, as listed in README of github repo.

C.2.4 Benchmarks. None.

C.3 Set-up

We provide a docker file in the github repo to build the docker image for all the following experiments, including both Aceso and baseline systems. Alternatively, you can also set up the environment by following the instructions in the "Set up environment" section of the github repo README.

C.4 Evaluation workflow

C.4.1 Artifact Functionality. We first demonstrate the functionality of Aceso with a small set of GPUs and compare its performance with the two baselines: Megatron-LM and Alpa. In this minimal working set, we will run the search and training step of Aceso, Megatron-LM and Alpa on the three small models: GPT-3 (1.3B), T5(770M) and Wide-ResNet(1B), on 4 GPUs. Please refer to the "Functionality check with small-scale experiments" section of github repo README for detailed instructions.

- E0 (Check functionality) [4 compute-hours]:
  [Execute] In the artifact repo, execute:

```
bash scripts/run_all_small.sh
```

  [Results] Check the training throughput and search cost:

```
python3 scripts/get_e2e_performance.py small
python3 scripts/get_search_cost.py small
```

  Check performance model accuracy:

```
python3 scripts/get_perf_model_acc.py small
```

C.4.2 Major Claims.

- C1: Aceso can find configurations with better performance than the baselines. For GPT-3 models, Aceso can achieve up to 1.27x speedup over Alpa, for Wide-ResNet, Aceso can achieve up to 1.33x speedup over Alpa. For T5 models, Aceso can achieve up to 1.50x speedup over Megatron-LM. This is proven by experiment (E1) described in Sec 5.1, whose results are illustrated in Figure 7.
- C2: Aceso only needs less than 5% search time of Alpa. This is proven by experiment (E2) described in Sec 5.1, whose results are illustrated in Figure 8.
- C3: Aceso can scale to the search of 1K-layer models. This is proven by experiment (E3) described in Sec 5.1, whose results are illustrated in Figure 9.
- C4: Aceso's performance model shows good prediction accuracy. This is proven by experiment (E4) described in Sec 5.2, whose results are illustrated in Figure 15 and Figure 16.

C.4.3 Experiments.

- E1 [3 compute-hours]: Run the search and training step of large-scale models selected in the paper evaluation. Please refer to the github repo README for detailed instructions in the "Reproducing results with large-scale experiments" section.
  [Preparation]
  – Prepare distributed training for Aceso and Megatron-LM: edit training scripts, provide environment variables `MASTER_ADDR`, `MASTER_PORT` and `NODE_RANK` as needed.
  – Prepare distributed training for Alpa: launch a Ray cluster.
  [Execute] Taking GPT-3 models as an example:
  – Run Aceso:

```
bash scripts/aceso_gpt_search.sh large
bash scripts/aceso_gpt_execute.sh large
```

  – Run Megatron-LM:

```
bash scripts/meagatron_gpt_search.sh large
bash scripts/meagatron_gpt_execute.sh large
```

  – Run Alpa:

```
bash scripts/alpa_gpt_search_execute.sh large
```

  [Results] Print out the results and plot Figure 7: (all the figures will be saved into `figures` folder)

```
python3 scripts/get_e2e_performance.py large
bash scripts/plot_fig7.sh
```

- E2 [1 human-minute]: Compare the search cost of Aceso and Alpa.
  [Results] Search cost has already been measured in E1, print out the summary of results and plot Figure 8:

```
python3 scripts/get_search_cost.py large
bash scripts/plot_fig8.sh
```

- E3 [7.5 compute-hours]: Search and execute Aceso, Megatron-LM and Alpa with 1K-layer model.
  [Execute]
  – Run Aceso:

```
bash scripts/aceso_gpt_search.sh scale
bash scripts/aceso_gpt_execute.sh scale
```

  – Run Megatron-LM:

```
bash scripts/megatron_gpt_search.sh scale
bash scripts/megatron_gpt_execute.sh scale
```

  – Run Alpa:

```
ray start --head
bash scripts/alpa_gpt_search_execute.sh scale
ray stop
```

  [Results] Print out the results and plot Figure 9:

```
python3 scripts/get_e2e_performance.py scale
python3 scripts/get_search_cost.py scale
bash scripts/plot_fig9.sh
```

- E4 [1 human-minute]: Compare the predicted and actual execution time and memory consumption.
  [Results] All the predicted and actual values are already collected in E1, you can print out the summary of results and plot Figure 15 and Figure 16:

```
python3 scripts/get_perf_model_acc.py large
bash scripts/plot_fig15.sh
bash scripts/plot_fig16.sh
```