Arena: Efficiently Training Large Models via Dynamic Scheduling and Adaptive Parallelism Co-Design

Chunyu Xue
dicardo@sjtu.edu.cn
Shanghai Jiao Tong University

Weihao Cui
weihao@sjtu.edu.cn
Shanghai Jiao Tong University

Quan Chen
chen-quan@cs.sjtu.edu.cn
Shanghai Jiao Tong University

Chen Chen
chen-chen@sjtu.edu.cn
Shanghai Jiao Tong University

Han Zhao
zhaohan_miven@sjtu.edu.cn
Shanghai Jiao Tong University

Shulai Zhang
zslzsl1998@sjtu.edu.cn
Shanghai Jiao Tong University

Linmei Wang
wanglm22@lenovo.com
Lenovo Research

Yan Li
yanli5@microsoft.com
Microsoft

Limin Xiao
xiaolm@lenovo.com
Lenovo Research

Weifeng Zhang
weifengz@lenovo.com
Lenovo Research

Jing Yang
jyang23@gzu.edu.cn
Guizhou University

Bingsheng He
hebs@comp.nus.edu.sg
National University of Singapore

Minyi Guo
guo-my@cs.sjtu.edu.cn
Shanghai Jiao Tong University

Abstract

Efficiently training large-scale models (LMs) in GPU clusters involves two separate avenues: inter-job dynamic scheduling and intra-job adaptive parallelism (AP). However, existing dynamic schedulers struggle with large-model scheduling due to the mismatch between static parallelism (SP)-aware scheduling and AP-based execution, leading to cluster inefficiencies such as degraded throughput and prolonged job queuing. This paper presents Arena, a large-model training system that co-designs dynamic scheduling and adaptive parallelism to achieve high cluster efficiency. To reduce scheduling costs while improving decision quality, Arena designs low-cost, disaggregated profiling and AP-tailored, load-aware performance estimation, while unifying them by sharding the joint scheduling-parallelism optimization space via a grid abstraction. Building on this, Arena dynamically schedules profiled jobs in elasticity and heterogeneity dimensions, and executes them using efficient AP with pruned search space. Evaluated on heterogeneous testbeds and production workloads, Arena reduces job completion time by up to 49.3% and improves cluster throughput by up to 1.60×.

EUROSYS '26, April 27–30, 2026, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3803571

CCS Concepts: • **Computing methodologies → Machine learning**; • **Computer systems organization → Cloud computing**.

Keywords: Large-scale model training, cluster scheduling

ACM Reference Format:
Chunyu Xue, Weihao Cui, Quan Chen, Chen Chen, Han Zhao, Shulai Zhang, Linmei Wang, Yan Li, Limin Xiao, Weifeng Zhang, Jing Yang, Bingsheng He, and Minyi Guo. 2026. Arena: Efficiently Training Large Models via Dynamic Scheduling and Adaptive Parallelism Co-Design. In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3767295.3803571

1 Introduction

Training jobs of large-scale models (LMs), such as GPT [25] and MoE [51], are vital workloads in production GPU clusters. With the rapid evolution of hardware architecture (e.g., NVIDIA Blackwell [17]), these clusters continually integrate new-generation GPUs with diverse compute capabilities and memory sizes, resulting in expanded cluster scale and heterogeneous resource types [40, 55, 81]. For instance, a production cluster [56] has thousands of GPUs with a mixture of NVIDIA H800 [20], A100 [16], and V100 GPUs [21].

To efficiently exploit these cluster resources, instead of user-specified rigid allocation [46], recent schedulers such as ElasticFlow [35] and Sia [45] *dynamically* (re)assign them to concurrent jobs, by adjusting the number ("elasticity") and type ("heterogeneity") of GPUs [26, 35, 42, 45, 53, 65, 68, 82]. From the job perspective, multiple parallelism dimensions,

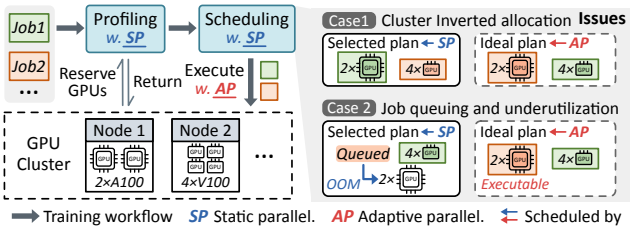

Figure 1. Workflow of dynamic scheduling based on static parallelism (SP) performance while executing with adaptive parallelism (AP). Our work targets optimizing this workflow.

including data [52], model [66, 70], and pipeline [64] parallelism, are used to execute LMs in a hybrid manner. Frameworks like Alpa [91] automatically explore the optimal hybrid parallelism plan under specific model and resource configurations [59, 77, 91], referred to as *adaptive parallelism* (AP) in this paper. Given the fixed model and resource allocation, it formulates a search space of parallelism plans and identifies the optimal plan via costly profiling [91].

However, prior dynamic schedulers [35, 45, 53, 65, 68] disregard adaptive parallelism when allocating resources, causing reduced cluster efficiency, as depicted in Figure 1. For training jobs, the scheduler allocates resources based on the performance of statically decided parallelism plans (or *static parallelism*, SP), typically data parallelism (DP), which is attained from model *profiling* [35, 42] or performance *estimation* [45, 65]. For example, ElasticFlow profiles jobs with DP across allocable resources in 10 minutes, Sia estimates multi-GPU DP throughput by linearly scaling single-GPU one with the GPU count. Given the dynamicity of AP (§2.2), such *mismatch between SP-aware scheduling and AP-based execution* leads to significant performance acquisition errors and degrades cluster efficiency. We empirically demonstrate this through two cases. First, the ideal resource allocation in the cluster could be "inverted" by the performance discrepancy between SP and AP execution (`Case1`). Second, the resource demands of LM jobs are overestimated (`Case2`), as static DP consumes the most memory among all parallelism. This results in the ideal plan being omitted, bringing more job queuing and resource underutilization in clusters.

Co-designing dynamic scheduling and adaptive parallelism is essential to mitigate these inefficiencies, though we find it challenging and lacking prior work. In this context, the scheduling and parallelism spaces form a joint optimization space via outer product (§3.2). A strawman approach is exhaustively profiling jobs with AP across allocable resources before scheduling, as in ElasticFlow and Lucid [42]. While intuitive, this incurs excessive time and hardware costs, as profiling needs reserving considerable GPUs for the time-consuming AP search (e.g., 20 minutes per allocable resource [91]). Given the prevalent GPU scarcity in production clusters [81, 89], this approach is impractical due to prolonged job queuing and resource contention. Another thought is analytically estimating job AP performance before scheduling, and executing jobs with AP at runtime, as in Sia and Gavel [65]. The execution is online profiled to refine estimates for better subsequent rescheduling. Despite its low cost, accurate estimation is non-trivial due to the shifting parallelism plan and scaling diminishing returns under dynamic resources. Inaccurate estimation undermines scheduling quality and cluster efficiency, which cannot be refined via frequent job rescheduling due to the high cost of AP search.

Our key insight to improve cluster efficiency and reduce scheduling costs is that, the scheduler can hierarchically unify low-cost profiling and AP-tailored estimation to efficiently navigate the joint scheduling-parallelism space. It is based on our observation that, given a model with specific resource, the performance relationship between two parallelism plans in AP search space can be analytically discerned without precise latencies, when the number of stages is fixed (§3.2). Building on this insight, we present **Arena**, a co-designed training system that dynamically schedules and efficiently executes large models with adaptive parallelism. Inspired by grid sampling [4], Arena shards the joint space into subspaces ("grids"). A grid of a job includes scheduling-parallelism plans with identical resource and *pipeline degree* (the number of stages). Within a grid, a near-optimal parallelism plan ("proxy plan") is analytically estimated; among grids, proxy plans are profiled and used for scheduling.

Concretely, for each grid, Arena identifies its proxy plan via roofline-based plan generation and Pareto frontier deduction in execution-free manner (§3.3). Given proxy plans, Arena profiles each via operator disaggregation and single-device profiling on fragmented resources (§3.4). With profiled jobs, Arena dynamically schedules them in elasticity and heterogeneity dimensions via job launching and scaling mechanisms, and a generalized event-driven policy with diverse objectives (§3.5). After scheduled, Arena executes jobs with efficient AP pruned by estimation and profiling results (§3.6). The main contributions of this work are:

- We reveal the issues of existing SP-aware schedulers in large-model scheduling, and identify the opportunity and challenges of joint scheduling-parallelism optimization.
- We propose Arena, a multi-model training system for large models, which co-designs dynamic scheduling and adaptive parallelism to improve cluster efficiency.
- We design low-cost, disaggregated profiling and AP-tailored, load-aware estimation, unified by a novel grid abstraction for efficient and precise job performance acquisition.

We implement Arena and evaluate it with various large models and production traces, in two real-world heterogeneous clusters with 64 and 384 GPUs of 2 types and a simulated cluster with 1,280 GPUs of 4 types. Extensive experiments show that Arena achieves up to 1.60× higher cluster

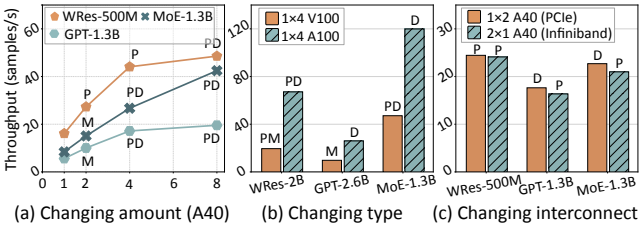

Figure 2. Benchmarking adaptive parallelism across various models and hardware. P/D/M denote pipeline, data and model parallelism. The optimal parallelism plan of AP is annotated.

throughput, reduces queuing delay by 74.9% and job completion time (JCT) by 49.3%, compared to four baselines. Arena is open-sourced at https://github.com/sjtu-epcc/arena.

2 Background and Motivation

2.1 Training Large Models in Clusters

Parallelism Strategies. Various parallelism strategies are proposed for multi-GPU model training. Data parallelism (DP) [52] splits batches across model replicas for concurrent computation, yet consumes substantial memory. Pipeline parallelism (PP) [43, 57, 64] groups operators into stages and splits a batch into microbatches, pipelining execution to improve throughput, but causes device idling. Model parallelism (MP), including tensor parallelism [66] and ZeRO [70, 71], splits operators or intermediate states across GPUs to reduce memory, at the cost of considerable communication.

Given the above pros and cons, the optimal parallelism plan is a hybrid combination that hinges on models, inputs, and hardware. To identify it, adaptive parallelism (AP) explores the search space of parallelism plans based on extensive profiling, by partitioning the model into pipeline stages, and parallelizing each across their assigned GPUs [59, 91]. Other strategies such as sequence parallelism [49] and expert parallelism [44] follow the paradigm of MP by splitting tensors across GPUs with communication incurred [59, 77].

Cluster Scheduling. Traditionally, resources are rigidly allocated on-demand, causing head-of-line blocking and resource fragmentation [53, 81]. To mitigate the issues in homogeneous clusters, recent schedulers [35, 42] dynamically adjust the number of GPUs for training jobs (**"elasticity"**).

As GPU manufacturers continue developing and rolling out new-generation GPUs (e.g., NVIDIA Hopper [20], Blackwell [17]), many production clusters continually integrate heterogeneous GPUs [40, 55, 56, 81]. Thus, both academic researches [26, 45, 65] and industrial practice [55, 56] put efforts on efficiently utilizing heterogeneous hardware, including dynamically adjusting GPU types for training jobs (**"heterogeneity"**). For instance, the clusters in [55, 56] consist of multiple regions, each for a job queue with a specific GPU type. Each cluster integrates a scheduler that allows users to specify "any-type" GPUs during job submission,

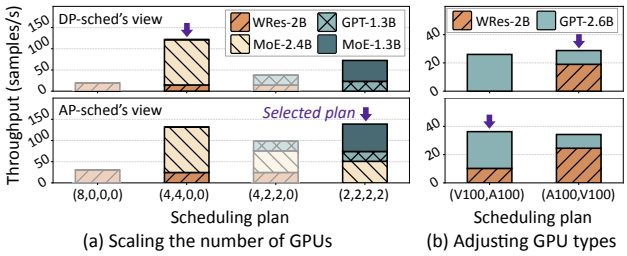

Figure 3. Case study of scheduling plan selection based on static data and adaptive parallelism. Missing bars indicate OOM issues. (a) `(a,b,c,d)`: a×A100 GPUs for WRes-2B, b× for MoE-2.4B, c× for GPT-1.3B, d× for MoE-1.3B. (b) `(A,B)`: $4 \times$ A GPUs for WRes-2B and $4 \times$ B for GPT-2.6B.

thereby reducing job queuing and improving resource utilization. Recent schedulers [45, 53] combine elasticity and heterogeneity dimensions for more flexible scheduling.

2.2 Problems in Scheduling Large Models

SP-aware scheduling in practice. Current schedulers, such as ElasticFlow [35], Sia [45], and EasyScale [53], disregard adaptive parallelism (AP) when allocating resources. As common practice, they assume that all training jobs are executed with static parallelism (typically DP[1]). This is reasonable in scheduling conventional models (e.g., ResNet, VGG), as they perform well with DP and rarely need AP. Another benefit is that the scheduler can acquire job performance via intuitive profiling [35, 42, 82] or estimation [45, 65, 68].

Mismatch between SP-aware scheduling and AP-based execution. However, the assumed static parallelism (SP) is mismatched with the actually used parallelism when LM jobs are executed with adaptive parallelism (AP), resulting in degraded cluster efficiency. This inefficiency primarily stems from the *performance discrepancies between SP and AP execution*. Figure 2 presents the AP performance of various models and hardware (configured as Table 1). We observe that instead of fixed parallelism patterns, AP exhibits significant dynamicity across different hardware. For instance, with sufficient resources, DP is preferred to improve computing concurrency (e.g., 8-GPU in Figure 2(a)); with limited memory, MP is used to reduce per-GPU memory footprint (e.g., GPT-2.6B in Figure 2(b)); with limited bandwidth, PP is selected for its lowest communication (e.g., MoE-1.3B in Figure 2(c)). Different models also exhibit varying parallelism preference on the same hardware. Consequently, determining the optimal plan in practice is a complex task.

As demonstrated, the optimal parallelism plan of AP and the throughput shift significantly across different configurations, rather than remaining static. In this case, assuming static parallelism such as DP covers only a marginal fraction

[1]Some schedulers like Sia [45] additionally support a fixed pipeline with manually partitioned stages instead of fully flexible AP.

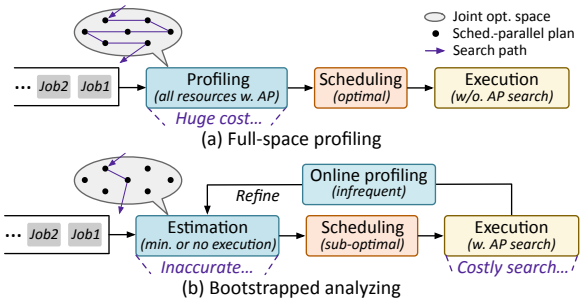

Figure 4. Workflows of two strawman approaches.

of the vast search space defined by AP, in which the optimal plan hinges on multiple factors such as models, inputs, and hardware. For this reason, SP-aware scheduling leads to performance acquisition errors when LMs are executed with AP. Figure 3 illustrates the impact of this mismatch on cluster efficiency by two frequently occurred cases, which is a quantitative case study of large-scale scheduling (§5.3).

***Case#1.* Cluster-level inverted allocation.** In Figure 3(a), with static DP, the throughput of plan `(4,4,0,0)` is higher than plan `(2,2,2,2)`, yet the latter is optimal with AP. Similar observations exist between plan `(A100,V100)` and plan `(V100,A100)` in Figure 3(b). Consequently, inverted allocation results in reduced overall throughput (e.g., 1.1×).

***Case#2.* Job-level prolonged queuing and underutilization.** Since DP consumes the most memory among all parallelism, MoE-2.4B is assigned 4 GPUs though trainable on 2 GPUs with AP (`(4,2,2,0)` in Figure 3(a)). This omits scheduling plans with "dense" allocation (high concurrency), causing prolonged job queuing and resource fragmentation. Moreover, overly allocating GPUs leads to underutilization and degraded throughput due to diminishing returns from excess resources [33, 35]. In Figure 2(a), the throughput of GPT-1.3B scales sub-linearly with the GPU count. Given 4 GPUs, using all for GPT-1.3B yields 1.5× lower throughput than using 2 GPUs each for GPT-1.3B and MoE-1.3B.

2.3 Deficiencies of Strawman Approaches

To improve cluster efficiency, the scheduler should consider adaptive parallelism (AP) performance for each job when allocating resources. In this context, the scheduling and parallelism spaces form a joint optimization space via outer product (§3.2). Given the limited prior work on this research topic, we in-depth analyze existing schedulers and identify two representative workflows: *full-space profiling* (e.g., ElasticFlow [35], Lucid [42]) and *bootstrapped analyzing* (e.g., Sia [45], Gavel [65]), as depicted in Figure 4. We use them as intuitive attempts to co-design scheduling and parallelism.

Full-space profiling results in huge time and hardware costs. The full-space profiling approach traverses the joint space before scheduling, i.e., profiling jobs with AP across all allocable resources (Figure 4(a)). At runtime, jobs are executed with the identified optimal plan. This ensures decision optimality and enables rescheduling without extra AP search, yet incurring excessive time and hardware costs beforehand.

Assume each job can be assigned up to N GPUs per type among M GPU types. Before scheduling, these NM GPUs should be reserved for model profiling. Let T_{ap} denotes the AP search cost (e.g., 20 minutes [91]), the profiling cost per job is $(N + (N-1) + ... + 1)MT_{ap} = O(N^2MT_{ap})$ GPU hours [72]. When K jobs arrive, say 8 from our trace statistics [40, 46, 81], with $N = 16$ and $M = 4$, the scheduler requires at least 1,500 GPU hours for profiling.

Such substantial cost renders full-space profiling impractical, making profiling cost reduction a key prerequisite in scheduling–parallelism co-design. Specifically, available GPUs are often scarce and fragmented in real-world clusters due to the prevalence of pending jobs with large resource demands [53, 81, 85, 89]. Jobs undergo much prolonged queuing to be allocated their maximal allocable resources (i.e., NM GPUs) for profiling. Moreover, substantial economic costs are incurred in this non-training phase (e.g., 5 dollars per GPU hour for AWS `pxde.24xlarge` with A100 GPUs [1]), while inter-job resource contention is exacerbated.

Bootstrapped analyzing results in inaccurate estimation. The bootstrapped analyzing approach estimates AP performance across allocable resources before scheduling (Figure 4(b)). At runtime, jobs are executed with AP and online profiled to refine estimates for better subsequent rescheduling. There are two major types of estimation. The first type analytically models the latency of computation operators by "FLOPs / {GPU peak performance × compute utilization}" [39, 67]. The utilization is empirically set by hardware profiling (e.g., 70% in [39]). This estimation becomes inaccurate under dynamic resources and adaptive parallelism due to the shifting compute and bandwidth utilization [50, 84, 91]. To illustrate this, we evaluate it using GPT-2.6B on 4 A40 GPUs, observing a 61.3% estimation error with 63.9% in compute (aligned with [86]) and 36.1% in communication latency ("volume / bandwidth").

The second type is partially profiling some configurations while estimating others (e.g., Sia [45], Centimani [84]). For instance, Sia assumes "throughput of 2-way DP is twice that of 1-way DP", profiling only 1-GPU throughput $\texttt{thr}_1$ per GPU type, and linearly estimating n-GPU throughput by $\texttt{thr}_n = \texttt{thr}_1 \times N_{gpu}$. It becomes inaccurate due to the overlook of parallelism shifting in adaptive parallelism and diminishing returns in resource scaling [33]. To illustrate this, we profile the AP throughput of GPT-1.3B from 1 to 16 GPUs, observing that the linear estimation yields errors from 1.14× (2 GPUs) to 2.12× (16 GPUs), with the optimal parallelism plan shifted from MP to DP and PP.

To investigate the impact of inaccurate estimation on scheduling quality, we add a knob η to regulate the fraction of precise profiling data in the linear estimation of Sia. For instance, when $\eta = 2$, we precisely profile 1 and 2-GPU

AP throughput ($\leq 2^{\eta-1}$). As η increases from 1 (original estimation) to 5 (fully precise data), the overall throughput improves by 1.19×, as configured in §5.3. However, the high cost of AP search hinders frequent correction of inaccurate estimation, ultimately degrading cluster efficiency.

2.4 Our Approach and Challenges

Analyzing the strawman approaches, we identify the *contradiction* in AP-aware scheduling: performant decisions require costly profiling, whereas swift estimation leads to poor scheduling quality. Prior works have focused on customizing scheduling algorithms, which cannot fundamentally resolve this contradiction [35, 45, 65]. In contrast, this work focuses on addressing this by designing principled approaches to enable efficient scheduling-parallelism co-design.

To reduce scheduling costs and ensure quality, we propose a two-fold approach that leverages the strengths of two strawmans to balance the "cost-accuracy" tradeoff, i.e., *unifying lightweight estimation and precise profiling*. Specifically, we find that estimation remains accurate in this context: for a model with specific resource, the relative performance between two parallelism plans in AP search space can be analytically discerned if sharing identical pipeline degree (§3.2). Beyond that, comparing scheduling-parallelism plans with varied pipeline degrees, resources, or models (inter-job) is essential for scheduling and requires profiling to obtain precise latency. There are three main challenges to develop and integrate this two-fold approach into cluster scheduling for efficient scheduling-parallelism co-design:

***C#1.* Lack precise parallelism estimation tailored for AP.** Coarse-grained methods fail to precisely estimate AP performance, as identifying optimal plans while adhering to AP optimality is non-trivial (§2.3). Instead of simplistic assumptions, a tailored, execution-free approach based on the above observation is needed to assess parallelism plans.

***C#2.* Lack practical profiling with minimal time and hardware costs.** Model-granularity profiling incurs high costs, prolonged queuing, and resource contention when AP is enabled (§2.3). Enhancing its production practicality requires reducing the number of occupied GPUs and the elapsed time while maintaining measurement accuracy.

***C#3.* Efficient scheduling and low-cost AP execution.** Given considerable number of jobs and heterogeneous resources, it is non-trivial to generate efficient scheduling plans with negligible overhead and high scalability. For fast deployment, it is essential to reduce the search cost of AP.

3 Arena Design

3.1 Overview

Arena is a multi-model training system for large-scale models, co-designing inter-job dynamic scheduling and intra-job adaptive parallelism (AP) to achieve high cluster efficiency, scalability, and production practicality. The key idea is to

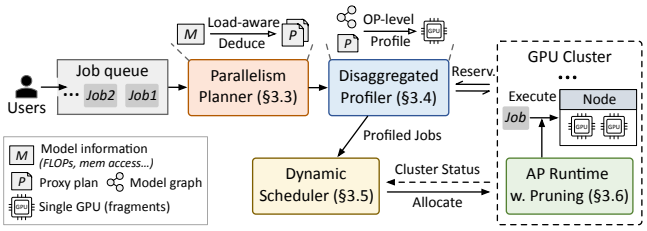

Figure 5. The overview of Arena architecture.

employ the two-fold approach (§2.4) to acquire the AP performance of jobs beforehand, dynamically schedule them in an AP-aware manner, and execute them with efficient AP.

To hierarchically unify estimation and profiling, Arena shards the joint optimization space into subspaces ("grids") based on the observation in §2.4. A grid of a job includes all scheduling-parallelism plans with identical allocated resource and pipeline degree. Within a grid, a near-optimal parallelism plan ("proxy plan") is analytically estimated; among grids, proxy plans are profiled and used for scheduling.

Figure 5 depicts the Arena architecture with four main components: planner, profiler, scheduler, and AP runtime to address the challenges (§2.4). Initially, users submit LMs to the job queue. Given a queuing job, *Parallelism Planner* (*C#1*) produces proxy plans via roofline-based plan generation and Pareto frontier deduction, in load-aware and execution-free manner (§3.3). For each proxy plan, *Disaggregated Profiler* (*C#2*) uses model graph for operator disaggregation and profiles non-redundant operators on fragmented resources (usually a single GPU) (§3.4). Given profiled jobs, *Dynamic Scheduler* (*C#3*) schedules in elasticity and heterogeneity dimensions via job launching and scaling mechanisms and a generalized event-driven policy (§3.5). At runtime, jobs are executed via *AP Runtime* with pruned search space (§3.6).

3.2 Sharding Space into Grids

Joint Optimization Space. In scheduling-parallelism co-design, the search space is formulated by the outer product of scheduling space $\mathcal{S}$ and adaptive parallelism space $\mathcal{P}$. $\mathcal{S}$ is defined as $\{(J_i, n, m)\}$, where the (J_i, n, m) scheduling plan indicates allocating job J_i with n GPUs of type m. The parallelism space $\mathcal{P}$ is given by $\{(P^{(s)}_{\text{inter}}, P^{(s)}_{\text{intra}})\}$ [91], where $P^{(s)}_{\text{inter}}$ groups operators into s stages and assigns GPUs for each, and $P^{(s)}_{\text{intra}}$ contains intra-stage parallelism for stages. Consequently, the joint space is formulated as:

$$\mathcal{J} = \mathcal{S} \times \mathcal{P} = \{(J_i, n, m, P^{(s)}_{\text{inter}}, P^{(s)}_{\text{intra}}) | s \in [1, n]\}. \quad (1)$$

Assume K jobs, O operators, N maximum allocable GPUs and M types, the overall complexity of $\mathcal{J}$ is $O(KNM\sum_s \binom{O}{s}\binom{N}{s}2^s)$. Such a huge optimization space consists of innumerable "scheduling-parallelism" plans, calling for an efficient approach to identify performant ones.

Granularity of Grid. For efficient exploration, the joint space is sharded into grids, i.e., $\mathcal{J} = \mathcal{J}_{\text{out}} \times \mathcal{J}_{\text{in}}$. Profiling is

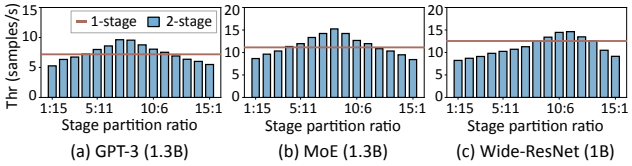

Figure 6. Throughput of various stage partition ratios compared to the single-stage case (with MP). `X:Y` represents the proportion of operators divided to the first and second stages. The later layers in `Wide-ResNet` are typically larger [87].

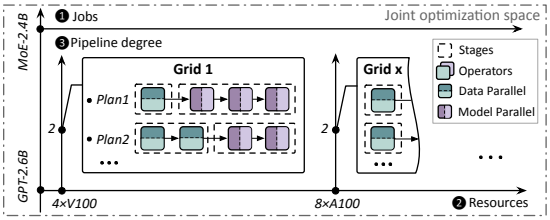

Figure 7. Sharding joint optimization space into grids.

applied in $\mathcal{J}_{\text{out}}$ (among grids), whereas estimation is used in $\mathcal{J}_{\text{in}}$ (in each grid). For instance, full-space profiling defines $\mathcal{J}_{\text{out}}$ as $\{(J_i, n, m, P^{(s)}_{\text{inter}}, P^{(s)}_{\text{intra}})\}$ and $\mathcal{J}_{\text{in}}$ as $\emptyset$; in Sia [45], $\mathcal{J}_{\text{out}}$ is given by $\{(J_i, m)\}$ and $\mathcal{J}_{\text{in}} = \{(n, P^{(s)}_{\text{inter}}, P^{(s)}_{\text{intra}})\}$.

To reconcile cost and accuracy, we determine the grid granularity based on the following observation. For a model with fixed resources, when adaptively parallelizing it *with a fixed pipeline degree*, a plan with balanced inter-stage loads consistently outperforms imbalanced ones on end-to-end performance (Figure 6). This is because each pipeline is bottlenecked by its slowest stage [38, 64]. An intuitive counter example to show the essentiality of fixing the pipeline degree is that, a model with a single stage always exhibits perfect inter-stage balance, yet may underperform multi-stage cases (e.g., up to 1.34× for GPT-3), as shown in Figure 6(a)-(c). Notably, training optimizations such as kernel fusion [29] and activation checkpointing [30] have limited impact on this observation, as they uniformly reduce kernel launch overhead or memory footprint across pipeline stages.

This observation serves as the criterion of assessing parallelism plans without precise latency measurements. Therefore, the grid of a job is defined as "optimization subspace with determined resource and pipeline degree" (Figure 7). Concretely, $\mathcal{P}$ is dissected into $\{(s) \times (O_s, D_s, P^{(s)}_{\text{intra}})\}$, where O_s and D_s are stage partition and GPU assignment for s stages. $\mathcal{J}_{\text{out}}$ becomes $\{(J_i, n, m, s)\}$ and $\mathcal{J}_{\text{in}}$ is $\{(O_s, D_s, P^{(s)}_{\text{intra}})\}$. This sharding reduces the profiling complexity to $O(KN^2M)$, while ensuring the optimality of AP-aware estimation.

3.3 Load-Aware Parallelism Planning

Instead of relying on extensive operator profiling before parallelization [64, 91], Arena builds an execution-free parallelism planner that assesses plans in each grid by inter-stage load balancing and intra-stage cost minimization.

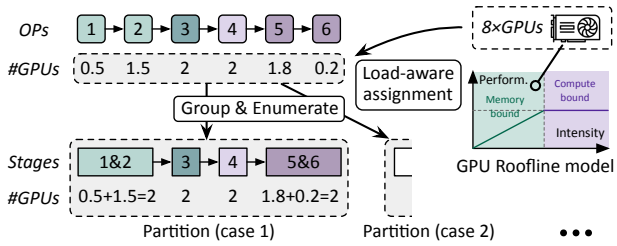

Figure 8. Example of stage partition and GPU assignment.

Roofline-Based Plan Generation. For a model with O clustered operators[2], Arena calculates operator FLOPs and memory access from static information (e.g., shapes). Since operator performance is collectively affected by these two factors [37, 86], we define its *load* as:

$$L_i = \frac{FLOPs_i}{R(I_i)} = \frac{FLOPs_i}{R(FLOPs_i/Mem_i)}, \tag{2}$$

where I_i is the arithmetic intensity of operator i, determined by input and parameter settings (e.g., batch size). $R(\cdot)$ is the roofline model function, which models the attainable performance on specific devices [10] and depends only on hardware specifications (e.g., SM count, memory bandwidth).

Arena then generates candidate stage partitions and GPU assignments. Since intra-stage parallelism evenly divides a stage across GPUs [66, 91], the load of a parallelized operator is L_i/d_i, where d_i is the GPU count. Thus, with the criterion of load balancing, Arena ensures *each stage's GPUs are proportional to their loads*. As shown in Figure 8, Arena calculates d_i for each operator by: $d_i = (L_i/\sum_{i'=1}^{O} L_{i'})N_{\text{gpu}}$, where N_{gpu} is the total number of GPUs. After that, Arena enumerates $\binom{O-1}{s-1}$ partitions by grouping operators into s stages, and computes GPU assignments by: $D_s^{(j)} = \sum_i d_i,\ \forall OP_i \in Stage_j, \forall j \in [1, s]$. In practice, $D_s^{(j)}$ is typically limited to a power-of-2 [45], thus Arena normalizes it by minimizing *computation bias* metric based on Euclidean distance [3]:

$$b_{\text{comp}} = (\sum_{j=1}^{s} |D^{(j)}_{s,norm} - D^{(j)}_s|^2)^{\frac{1}{2}}, \tag{3}$$

where $D^{(j)}_{s,norm}$ is the normalized GPU count for stage j. This metric quantifies the gap between the actual and theoretically optimal assignments. A larger value indicates a more imbalanced pipeline with poorer end-to-end performance.

To generate parallelism plans, Arena further determines intra-stage parallelism per stage by minimizing communication cost within memory limits. The observation is that, when evenly parallelizing stages, different intra-stage parallelism incurs varying communication and memory costs. While sharding dimensions (e.g., batch dimension for DP) affect tensor layouts, communication dominates the selection with much larger latency impact (e.g., 6.27ms v.s. 0.03ms for

[2] In practice, operators (e.g., matmul) are pre-clustered to control problem size (e.g., $O = 16$ [91]). We use the term "operator" to represent this.

a GEMM with $TP = 4/DP = 4$), as compute (SM) utilization is saturated under large-scale training workloads [77, 91].

Deducing Pareto Frontier and Proxy Plan. Without execution, it is infeasible to define a unified metric (e.g., end-to-end latency [48, 91]) that considers both computation and communication, hindering direct comparison between parallelism plans. Therefore, Arena evaluates computation and communication separately. The computation performance is assessed by b_{comp}, which quantifies the impact of stage partitioning and GPU assignment. For communication, we introduce a *communication load* metric based on pipeline patterns (§3.4) to evaluate the end-to-end cost:

$$l_{\text{comm}} = (B-1) \cdot \max_{1 \le j \le s} \{ \sum_{i} c_i \} + \sum_{i'=1}^{O'} c_{i'}, \quad \forall OP'_i \in Stage_j, \quad (4)$$

where B is the number of microbatches, OP'_i denotes communication operator i and c_i is the cost. There are totally O' communication operators. This metric models sequentially executed intra-stage communication (e.g., all-reduce), while excluding P2P communication as it is much smaller and usually overlapped with stage computation.

With the two metrics, Arena frames plan comparison as a multi-objective optimization problem [6, 23]: a parallelism plan is considered superior if it achieves both lower b_{comp} and l_{comm} compared to another plan. Thus, Arena minimizes two metrics simultaneously, deducing a Pareto frontier by traversing parallelism plans and collecting *non-dominated* ones (no other plan outperforms them on both metrics). If the collected plans exceeds a threshold, the Pareto frontier is reduced by dropping the plan with higher communication load from the pair with the most similar stage partition.

Given the Pareto frontier, Arena identifies the proxy plan from non-dominated plans. Since computation typically dominates end-to-end performance [39, 91], Arena filters plans with the minimum computation bias and then selects the plan with the lowest communication load, which achieves 93.4% of the optimal plan performance on average (§5.4).

3.4 Disaggregated Profiling

Though the grid sharding of Arena significantly reduces profiling complexity (§3.2), considerable resources are still required to profile proxy plans, especially for LMs. This incurs job queuing and other issues as analyzed in §2.3. To address this, we find that the operator execution after compilation remains sequential and isolated despite data dependencies [59, 91], thereby enabling operator-level profiling. Existing solutions include zero-GPU blackbox prediction [50, 86] with extensive pre-fitting and unstable errors on dynamic resources, and multi-GPU measurement [32, 84] that requires full GPU access to benchmark operator latencies.

In contrast, Arena designs a profiler for both precise and efficient profiling on fragmented GPUs (e.g., a single GPU), while generalizing to various runtime optimizations [29]. In this subsection, we detail why and how Arena disaggregates computation and communication operators, then present the profiling approach using a single device.

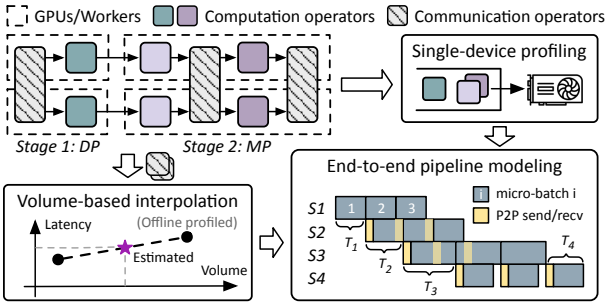

Figure 9. Workflow of single-device disaggregated profiling.

Operator Disaggregation. We observe *compute redundancy* when profiling parallelized LMs, including: (i) Workers of the same stage execute identical operators [59, 91], while the stage latency equals that of a single worker; (ii) Many LMs contain a large proportion of repeated operators with identical latency [25, 76]; (iii) Device stalls (e.g., pipeline bubbles [43]) largely exist during model execution.

To eliminate unnecessary computations and device stalls, Arena disaggregates the operators of a model. Specifically, Arena pre-compiles the functions of each parallelized stage and generates intermediate representation (IR) objects. Then, Arena extracts computation graphs from IR objects and retrieves operator configurations (e.g., shapes, data types) from them. The disaggregated computation and communication operators are taken into separated consideration:

***1)* Computation operators.** The latency of a computation operator is affected by multiple factors. First, different kernel implementations with irregular latencies are launched for diverse shapes, data types, and GPU architectures [24, 86]. Second, various runtime optimizations (e.g., operator fusion) are adopted to optimize execution [29, 61, 73]. Thus, online profiling is essential to obtain the precise latency of computation operators, and can be done on a single GPU.

***2)* Communication operators.** The latency of a communication operator depends on primitive types, network topologies (fixed after hardware setup), and transfer volume. Advanced optimizations such as primitive substitution and data compression also statically rely on primitive types and GPU interconnection [27, 54, 91]. Thus, under the same primitive and network topology, the latency of a communication operator is proportional to data transfer volume [2, 37, 39].

Single-Device Profiling. Arena dispatches computation and communication operators to dedicated modules (Figure 9). For computation, Arena reconfigures operator shapes based on the intra-stage parallelism (e.g., dividing batch dimension by 2 for 2-way DP). For each stage, Arena constructs a single-device executable with reconfigured operators, corresponded to a worker of that stage. Lastly, Arena profiles

executables with the same runtime optimizations as in direct execution (thus achieving kernel-level equivalence), and captures the latencies of launched kernels [19]. Operators with identical configurations are not profiled repeatedly.

For communication, Arena offline samples representative data volumes and profiles candidate primitives across pre-accessible hardware (e.g., 2×4 A100 nodes with Infiniband). At online, Arena estimates operator latency by interpolating based on the transfer volume under the same primitive and hardware. The stage latency is calculated by summing the latencies of all operators due to sequential execution.

Finally, Arena models the execution schedule of specific pipeline (e.g., 1F1B [64]) and computes the end-to-end latency T_{e2e}. As shown in the bottom right of Figure 9, T_{e2e} consists of: (i) Latencies of the first microbatch for all stages ($T_1 + T_2 + T_3 + T_4$); (ii) Latencies of the remaining $B-1$ microbatches for the slowest stage ($(B-1) \times (T_3 - T_{\text{p2p}})$), where computation-communication overlap is modeled.

3.5 Dynamic Scheduling

To query the AP performance of a job on specific resources, Arena traverses relevant grids for the best-performing one, using its profiled data as inputs to the scheduler.

Mechanisms. Arena provides two fundamental "operations" during job launching and training phases.

***1)* Priority-Based Multi-Queue Launching.** In Arena, jobs are assigned priority $\lambda \in [1, P]$ based on user or vendor specifications (aligned with production practice [41, 83]). Arena maintains P queues to schedule job launching order (one for each λ value). A smaller λ indicates higher launching priority, while jobs within the same queue are launched in enqueue order. A launched job adheres to the "run-to-completion" rule, i.e., cannot be preempted or scaled to a non-executable state due to high migration overhead of large models. To mitigate starvation caused by head-of-line blocking [46, 83], when a job blocks λ-queue due to substantial resource demands, a subsequent job in λ-queue is allowed to preempt resources (when feasible) for earlier launch — a privilege not for jobs in $(\lambda+1)$-queue. For fairness, a job priority λ is promoted to $(\lambda-1)$ (if $\lambda > 1$) after prolonged queuing. The selection of P value is analyzed in §5.8.

***2)* Two-Dimensional Scaling.** To improve cluster efficiency and reduce fragmentation, Arena lively *scales* resources for both newly launched and in-flight jobs, by adjusting the allocated number ("elasticity") and type ("heterogeneity") of GPUs. To ensure job locality (or rack affinity), Arena follows the buddy allocation rule and attempts periodic job migration for defragmentation [35, 89]. User-specified hyperparameters (e.g., global batch size) remain fixed to ensure that the scaling does not compromise model quality (§5.8).

Notably, Arena employs dynamic GPU type switching to reduce job queuing and improve resource utilization, while ensuring intra-job homogeneity to guarantee training efficiency and ease of management [45, 53, 65]. This is because production clusters typically house homogeneous GPUs in the same region with neighboring nodes. Allocating heterogeneous GPUs to a single job results in cross-region communication with much limited bandwidth [31, 40, 55]. Moreover, due to the risks of compute stragglers and resource fragmentation, intra-job heterogeneity is more considered in cloud-based training, where large-block GPU allocation in a single region is often infeasible [28, 74].

Algorithm 1 Generalized Event-Driven Job Scheduling

```
1:  Global: In-flight jobs J_L, priority queue Q, cluster resources R, search depth D
2:  function Schedule(sEvents, cEvents)                    ▷ Entrypoint
3:      R ← Free(cEvents.resources)                        ▷ Free resources
4:      J_L ← J_L − cEvents.jobs
5:      Q.enqueue(sEvents); plans ← ∅; λ' ← P + 1
6:      for E ∈ Q.sort() do
7:          job ← E.job; plans[job] ← LEventHandler(E)
8:          if plans[job] is ∅ and job.λ ≤ λ' then λ' ← job.λ
9:          if plans[job] is ∅ and job.λ > λ' then break
10:     plans ← plans+ InFlightHandler()
11:     Finalize(plans, J_L, R)                            ▷ Allocate resources
12: function LEventHandler(Event)                          ▷ Launch a job
13:     while not Event.alloc and plan.len < D do
14:         plan ← GetOptimalScaleDown(J_L + Event.job, R)
15:     if Event.alloc then J_L ← J_L + Event.job; Q.dequeue(Event)
16:     return plan if Event.alloc else ∅
17: function InFlightHandler()                             ▷ Scale up in-flight jobs
18:     while job* is not ∅ and plan.len < D do
19:         job*, OP* ← GetOptimalScaleUp(J_L, R)
20:         if job* is not ∅ then plan ← plan+ < job*, OP* >
```

Generalized Event-Driven Scheduling. Arena employs a generalized event-driven policy with support of diverse scheduling objectives. For instance, the throughput maximization objective is formulated as:

$$\max_{A} \sum_{i=1}^{K} \left(\sum_{m=1}^{M} \sum_{n=1}^{N} A_i[m,n] \times Thr_i^{m,n} \right), \tag{5}$$

where $A_i[m,n] = 1$ denotes allocating job J_i with n GPUs of type m. $Thr_i^{m,n}$ is the AP throughput. Other intuitive constraints are not shown due to space limits. For deadline awareness, Arena adds an additional constraint:

$$\forall i, \sum_{m=1}^{M} \sum_{n=1}^{N} \frac{A_i[m,n] \times B_i N_i^{iter}}{Thr_i^{m,n}} \leq DDL_i, \tag{6}$$

where B_i, N_i^{iter}, and DDL_i denote the global batch size, remained iteration count, and deadline of job J_i.

Algorithm 1 outlines the scheduling workflow of Arena with three job event types: submission (`sEvent`), launch (`lEvent`), and completion (`cEvent`). In each scheduling step, Arena invokes `Schedule` to handle pending `sEvents` and `cEvents` (line 2-11). For `lEvent`, Arena calls `LEventHandler`

to iteratively locate the optimal job to scale down based on the scheduling objective (line 12-18). If idle resources exist, Arena then calls `InFlightHandler` to iteratively locate the optimal in-flight job to scale up (line 19-24).

To efficiently locate the optimal job for scaling, Arena heuristically identifies the job that yields the greatest benefit after scaling. This is backed by the observation in §2.2 that for different models and hardware, scaling operations lead to varied performance variations with diminishing returns. Therefore, Arena scales down jobs with excessive resources but limited performance when resources are insufficient, and scales up promising jobs if idle resources exist. Prior schedulers have demonstrated the near-optimality of such heuristics [35, 53]. To reduce scheduling overhead, a parameter named *search depth* limits the maximum iterations in the iterative scaling process (evaluated in §5.8).

Extensibility. Beyond the throughput maximization and deadline awareness, Arena is compatible to other objectives that allocate resources based on job performance. For example, the Finish-Time Fairness [62] can be formulated as:

$$\min \max_{1 \le i \le K} \left(\sum_{m=1}^{M} \sum_{n=1}^{N} \frac{N_i^{iter} T_i^{m,n}}{A_i[m,n] \times Q^{m,n}} \right), \tag{7}$$

where $T_i^{m,n}$ is the iteration time and $Q^{m,n}$ measures the allocated resources. In addition, the scheduling policy can be modified to seamlessly integrate with other algorithms, such as solving Equation 5 by ILP optimization.

3.6 Runtime Pruning

At runtime, Arena executes scheduled jobs with adaptive parallelism (AP), pruning the search space via three predefined rules for fast deployment. First, for each job with allocated resources, Arena selects the best-performing grid (G_b) and directly applies its pipeline degree. Second, Arena identifies the most imbalanced stage partition in the Pareto-optimal plans of G_b, pruning plans with greater imbalance. Third, for each stage, if its operator composition matches that of a stage in certain Pareto-optimal plans, its assigned GPU count and intra-stage parallelism are directly determined. These rules are compatible with various AP frameworks (e.g., Alpa [91], Aceso [60], nnScaler [59]). Alternatively, Arena supports directly using the proxy plan for zero-overhead execution. Experiments in §5.4 demonstrate 5.48× search cost reduction (96.2% of the optimal performance) on average.

4 Implementation

We implement Arena with 13K LoC in Python: 2,800 lines for the parallelism planner, 5,600 lines for the disaggregated profiler, and 5,200 lines for the scheduler. Arena provides a simulator to conduct large-scale scheduling experiments, ensuring high fidelity by sharing scheduling codes and logics with the real-testbed scheduler. We build the training backend based on Alpa [91] in JAX framework [5], with minor modification of 500 LoC on its `stage_construction` and `stage_profiling` modules to support parallelism pruning at runtime. Besides such modifications, Arena modules remain independent of the underlying backend.

Table 1. Specifications of heterogeneous GPU clusters in evaluation. GPUs with † are equipped with NVLink [22].

GPU	Arch.	Mem(GB)	Interconnect	#GPU/Node
H100†	Hopper [20]	80	NV. ConnectX-6 [8]	8
L20	Ada [15]	48	NV. ConnectX-6	16
A100†	Ampere [16]	40	NV. ConnectX-5 [7]	4
A40	Ampere	48	NV. ConnectX-5	2
A10	Ampere	24	NV. ConnectX-6	2
V100†	Volta [21]	32	NV. ConnectX-5	16

For parallelism planning, we obtain operator information (e.g., FLOPs, memory access) via static model analysis based on XLA [11] and its HLO IR. We borrow the ILP solver of Alpa to minimize communication costs in intra-stage parallelization phase. For disaggregated profiling, stage precompilation is implemented based on XLA. We implement a kernel profiler with 500 LoC in C++ based on NVIDIA CUPTI [19], using it to capture kernel latencies during single-device profiling. We offline profile representative communication primitives (e.g., all-reduce) based on XLA, NCCL [18] and Ray [63]. For scheduling, we use gRPC [13] to communicate between the scheduler and distributed training processes. Arena schedules job launch and completion events every 5 minutes and inspects cluster status every 20 seconds. Arena implements job migration operations based on checkpoint-resume [82], while remaining extensible to advanced state management techniques [79, 80], such as delegating model and dataset states to Tenplex [79] for faster model migration and reinitialization.

5 Evaluation

5.1 Experimental Setup

Physical Testbeds. We use two heterogeneous clusters: (i) *Cluster-A*: A cluster of 32 nodes and 64 GPUs. 16 nodes have Intel Xeon Gold 5318Y CPUs and 2 NVIDIA A40 GPUs; other 16 nodes have the same CPUs but with 2 A10 GPUs. (ii) *Cluster-B*: A cutting-edge cluster with 128 NVIDIA H100 GPUs (16 nodes, each with 2 Intel Xeon Platinum 8457C CPUs and 900GB/s NVLink) and 256 L20 GPUs (16 nodes, each with the same CPUs and 64GB/s PCIe4.0 connection).

Simulated Cluster with Higher Heterogeneity. The simulated cluster has 1,280 GPUs of 4 types: A100 (80 nodes), A40 (160 nodes), A10 (160 nodes), and V100 (20 nodes), as detailed in Table 1. For simulation, we extensively pre-measure AP performance across diverse models, hyperparameters, and hardware, replacing real model training with process sleeping and querying pre-measured data. The measurement naturally includes the impact of network topologies (e.g.,

Table 2. Model configurations used in the experiments.

Model	Global Batch Size	#Params (billion)
Wide-ResNet [87]	[256, 512, 1024]	[0.5, 1.0, 2.0, 4.0, 6.8]
GPT-3 [25]	[128, 256, 512]	[0.76, 1.3, 2.6, 6.7]
GShard MoE [51]	[256, 512, 1024]	[0.69, 1.3, 2.4, 10, 27]

NVLink/Infiniband within/across A100 nodes). We model the overhead of full-space/space-pruned AP and checkpoint-resume by pre-measuring across model and hardware scales.

Models and Traces. We conduct experiments using three LMs in Table 2. For GPT-3 and MoE, we use the sequence length of 1024. The number of micro-batches is set to 4× the number of pipeline stages [43]. We use three production traces to evaluate the scheduling performance, including a two-week Philly trace with over 13,000 jobs [46], a Helios Venus trace [40], and a PAI trace [81]. For each job record (ID, submission time, duration), we randomly generate GPU count, type, model configurations, and iteration count to adapt the traces to heterogeneous scenarios.

Baselines. We compare Arena against four baselines: (i) **FCFS** [14] rigidly schedules jobs with user-specified resources in arrival order. (ii) **Gavel** [65] performs heterogeneity-aware scheduling by formulating various objectives (e.g., throughput maximization) into ILP-based optimization problems and solving them. (iii) **ElasticFlow** (EF) [35] heuristically schedules jobs by scaling their GPU counts with deadline awareness and throughput maximization in a homogeneous cluster. (iv) **Sia** [45] schedules statically parallelized jobs, adjusting GPU counts, types, and hyperparameters by solving an ILP-based goodput maximization problem. Other RL-based schedulers [69] are not used as their performance upper bounds would not exceed our ILP-based baselines.

Job Execution and Profiling. For baselines designed for SP-aware scheduling, we enable adaptive parallelism (AP) in job execution. Once admitted, a job cannot be suspended by others. Moreover, in physical and simulated testbeds, all dynamic schedulers require profiling results to schedule jobs. We follow [35, 42, 65] to conduct all profiling ahead-of-time due to budget constraints, including full-GPU DP profiling for baselines, bootstrapped DP profiling for Sia, and grid-based techniques for Arena. We prepend corresponded profiling overheads in scheduling experiments, which are directly measured in ahead-of-time profiling across models and hardware scales. The effectiveness of the Arena planner and profiler is exclusively evaluated in §5.4 and §5.5.

5.2 Evaluation in Real-World Clusters

We first evaluate Arena on two heterogeneous testbeds (§5.1) using a 6-hour trace of 244 jobs from Philly trace [46]. For *Cluster-B* that is equipped with newer-generation GPUs, we scale up the workload by increasing model sizes and number of iterations (by 10× on average) in the job trace.

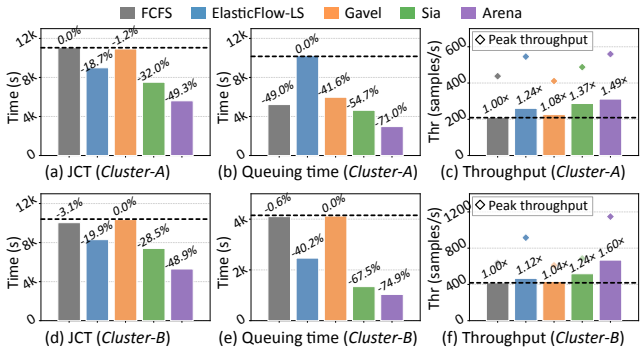

Figure 10. Performance on two real-world testbeds. -LS denotes ElasticFlow uses loosened job deadline [35].

In *Cluster-A* (Figure 10(a)-(c)), Arena improves average cluster throughput by up to 1.49×, reduces average JCT of all submitted jobs by 49.3% and average queuing time by 71.0%. The benefits stem from two main reasons. First, Arena identifies and profiles near-optimal parallelism plans for AP-aware job scheduling, achieving wiser resource allocation (e.g., mitigating overestimation issues in §2.2) and inter-job adjustment (e.g., allocating more GPUs to those with higher performance potential). Second, Arena reduces the time and hardware costs for job profiling and conditionally preempts blocked jobs, bringing better resource utilization for training and more execution opportunities.

In *Cluster-B* with more advanced GPUs and heavier workloads (Figure 10(d)-(f)), Arena also achieves significant improvements: up to 1.60× higher cluster throughput, 47.3% JCT reduction, and 74.9% queuing time reduction. Similar to the results in *Cluster-A*, the awareness of AP performance in job scheduling acts as the primary reason for the benefits. For baselines, though using more and better GPUs (thereby reducing JCT and queuing time), profiling larger models using DP with replicated parameters exacerbates the resource quota overestimation for AP execution. Moreover, the enlarged cluster size also leads to larger scheduling space with increased resource misallocation risks.

Simulation Fidelity. We conduct simulations using the same configurations as real-world experiments in *Cluster-A*. The average simulation error across Arena and baselines is 3.16% for throughput and 7.22% for JCT, which clearly confirms the fidelity of our forthcoming large-scale simulations.

5.3 Larger-Scale Simulations

Figure 11 illustrates the cluster throughput of Arena and the baselines with the one-week Philly trace. In the first three days, the cluster load is low with several transient bursts; in last four days, the cluster experiences intensive heavy loads. Arena outperforms baselines when scaling up under burst loads because of better scheduling plans with more admitted jobs (❶). When loads decrease, Arena scales down earlier as jobs are efficiently completed (❷).

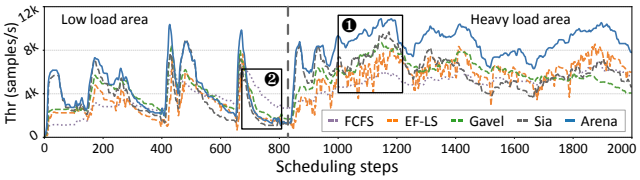

Figure 11. In-depth analysis of cluster throughput in 1,280-GPU simulated cluster. The scheduling interval is 5 minutes.

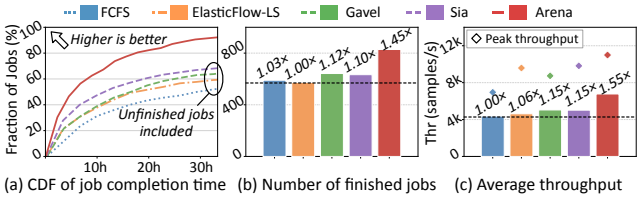

Figure 12. Performance analysis in the simulated cluster.

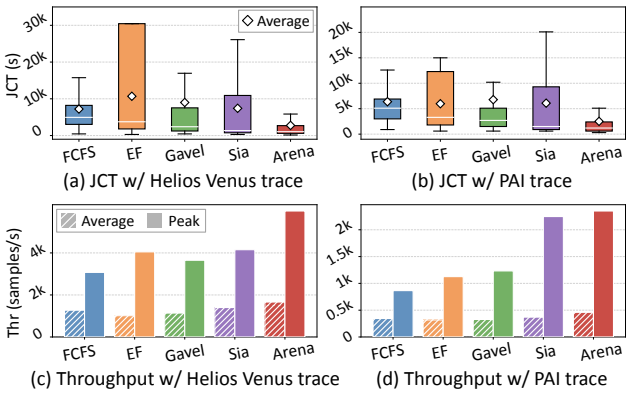

Figure 13. Performance on Helios (moderate loads) and PAI trace (light loads). Other settings are the same as before.

Figure 12 further shows numerical comparisons of the above experiment. As observed, Arena reduces average JCT by 81.3% (FCFS), 80.5% (ElasticFlow-LS), 76.6% (Gavel) and 75.2% (Sia), completing up to 1.45× more jobs. From the cluster perspective, Arena outperforms baselines with up to 1.55× higher average throughput and 1.58× higher peak throughput. Notably, the average job rescheduling count in Arena is only 2.29, as Arena schedules based on precise AP performance and limits the search depth of job scaling (3 in simulation). Gavel outperforms ElasticFlow on JCT and throughput, contrary to real-testbed results due to higher heterogeneity of the simulated testbed (§5.7). Sia outperforms other baselines under low loads, yet is throttled when cluster load bursts (Figure 11), as it schedules with DP and overestimates the resource demands of jobs.

Evaluation with Other Traces. We use a one-day trace with moderate load from Helios [40] and another with low load from PAI [81]. Figure 13 shows that Arena consistently outperforms baselines, achieving up to 74.2% and 63.0% JCT reduction, as well as 1.64× and 1.44× average throughput improvements on Helios and PAI traces. These improvements are consistent with the results of Philly trace experiments.

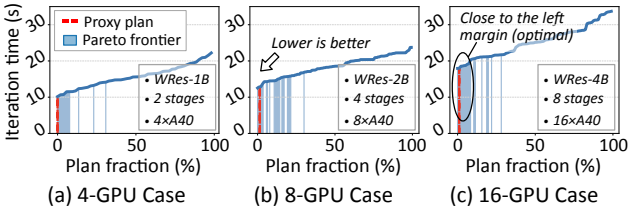

Figure 14. In-depth case study of Pareto frontier deduction.

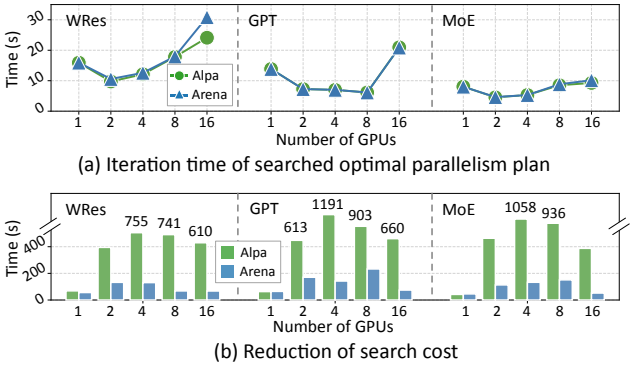

Figure 15. Performance of AP with pruning.

5.4 Effectiveness of Parallelism Planning

Pareto Frontier and Proxy Plan Deduction. We evaluate the optimality of parallelism deduction by three case studies (Figure 14). Due to space limits, all experiments employ Wide-ResNet of varying sizes (most complex layer structure). To build performance curves, we enumerate, profile, and sort all parallelism plans by iteration time in ascending order.

In each grid, the proxy plan achieves 86.2%, 85.6%, and 94.3% of the optimal performance on 4, 8, and 16 GPUs. Extending to more grids, the proxy plan matches the optimal plan in 86.5% cases, with average performance gap of 9.82% in mismatched cases. Moreover, across various models and hardware, the best proxy plan (used for scheduling) achieves average 93.4% performance of the AP searched optimal plan.

Parallelism Pruning. We further evaluate the effectiveness of parallelism pruning against Alpa [91], as shown in Figure 15. The optimal parallelism plan searched by Arena achieves 96.2% of Alpa performance on average, benefited from the optimality of Pareto frontier. With pruning, Arena reduces the AP search cost by 5.48× on average and up to 10.88×, owing to the efficient pruning rules as in §3.6.

5.5 Efficiency of Disaggregated Profiling

Figure 16 evaluates Arena profiler in terms of error rate (i.e., the gap between profiled and directly measured latencies) and profiling cost reduction. Through precise operator profiling and end-to-end modeling, Arena achieves average error

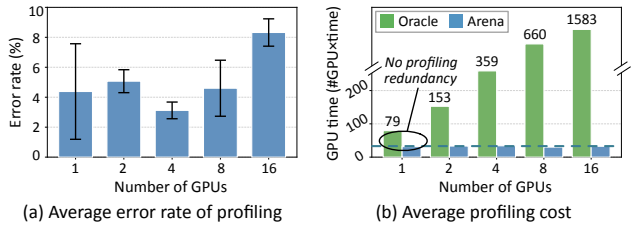

Figure 16. Performance of disaggregated profiling. (a) Each bar shows the average error across diverse models and hardware. (b) `Oracle` represents directly executing the model.

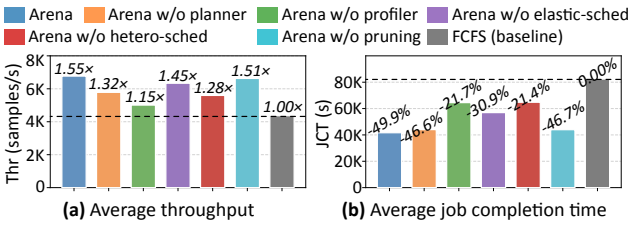

Figure 17. Performance breakdown of Arena.

rates of 4.4%, 5.1%, 3.1%, 4.6%, and 8.3% for 1, 2, 4, 8, and 16 GPU cases, sufficient for job scheduling (Figure 16(a)). Moreover, Arena reduces the GPU time (i.e., elapsed time × occupied GPU count [62]) by 18.1× on average and 2.55× at least, as compared to direct measurement. This stems from eliminating redundant computations and device stalls, while avoiding online communication profiling that remains time-consuming under high traffic or limited bandwidth.

5.6 Deadline-Aware Scheduling

We also evaluate the generality of Arena for other objectives by enabling deadline awareness and comparing with ElasticFlow [35]. Arena ensures strict deadlines while optimizing throughput, dropping jobs that cannot meet deadlines. Compared with ElasticFlow, Arena improves *deadline satisfaction ratio* (the proportion of jobs satisfying their deadlines) by 1.69× and reduces JCT by 26.1%. Moreover, Arena achieves 1.73× higher average throughput and 1.96× higher peak throughput. This is attributed to the AP-aware scheduling of Arena, which enables efficient resource allocation with more jobs being admitted and efficiently executed.

5.7 Ablation Studies

Performance Breakdown. To understand the sources of the benefits, we evaluate Arena by sequentially disabling each component (Figure 17). As the two key components of Arena, the profiler and planner have notable impact on cluster throughput. Disabling the profiler results in 25.8% less throughput and 56.3% higher JCT due to aggravated resource contention between to-profile and in-flight jobs (§2.3). Without planner, the scheduler employs inaccurate performance data when assuming jobs are executed with DP, resulting in 14.8% less throughput and 6.59% higher JCT.

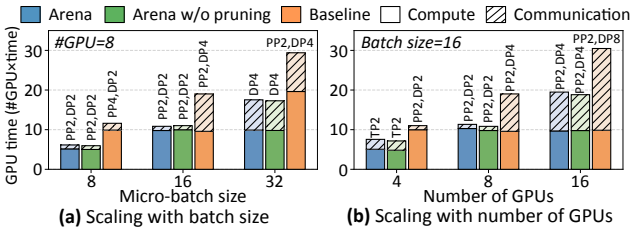

Figure 18. Training time breakdown of GPT-2.6B with A40 GPUs. The searched optimal plans are marked above bars.

Another dominant component is the heterogeneity-aware scheduling, as `w/o hetero-sched` reduces cluster throughput by 17.4% and increases JCT by 56.9%. This impact surpasses `w/o elastic-sched` due to the high heterogeneity of the simulated testbed, which explains why Gavel outperforms ElasticFlow in simulation (4 GPU types) but falls short in real-world testbeds (2 GPU types). Moreover, `w/o pruning` exhibits limited performance decrease due to the limited per-job rescheduling events (2.29 in §5.3).

Job-Level Breakdown. To delve into the performance gains at the job level, we breakdown iteration time into latencies of compute (e.g., GEMM) and communication kernels (e.g., all-reduce) to assess Arena, Arena w/o pruning, and Sia. Givan that Sia overestimates job resources (§2.2), for ablation, we statically assume 2× more GPUs allocated by it. As shown in Figure 18, Arena generates the same optimal plans with full-space AP (<5% performance gap), aligned with results in §5.4. Compared to Sia, Arena reduces 41.2% GPU time on average by efficiently parallelizing on less GPUs and alleviating diminishing returns in resource scaling. For example, increasing DP has negligible impact on compute GPU time, yet largely increases communication GPU time by up to 9.15× due to cross-node all-reduce operations. Consequently, the baseline allocates more GPUs for the job but fails to proportionately improve its training efficiency.

Effectiveness of Dynamic Scheduling. We study the effectiveness of Arena scheduler by disabling other components and scheduling based on job DP performance (aligned with baselines). We scale job lifespan (iteration count) of the trace in §5.3 for evaluation under varying workloads (Figure 19). As job lifespan increases, the benefits of Arena become more significant with up to 1.59× higher average throughput, as it efficiently schedules jobs via priority-based launching, 2D scaling, and the event-driven, throughput maximization policy. With sparse jobs (sufficient idle resources), however, the benefit is limited, as the multi-level job queues of Arena fall back to FCFS and no job down-scaling is needed.

Homogeneous Scheduling Study. To illustrate the robustness of the scheduling-parallelism co-design, we further assess Arena using a homogeneous real-world testbed with 128 H100 GPUs in *Cluster-B*. Evaluated with the same trace in §5.2 (all jobs are reassigned with only H100 GPUs), Arena

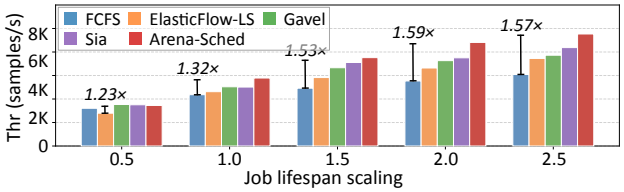

Figure 19. Effectiveness of Arena scheduler (Arena-Sched) over job lifespan scaling, other components are disabled.

improves average cluster throughput by 2.36× (FCFS), 1.48× (ElasticFlow-LS), 2.04× (Gavel), and 1.21× (Sia). These results demonstrate the benefits of Arena gained from AP-aware scheduling, regardless of underlying hardware setups.

5.8 Sensitivity Analysis

Impact of Parameters. We evaluate the impact of two parameters. A smaller number of priority queues P improves managing efficiency and reduces starvation, while a larger P enhances fairness for high-priority, resource-intensive jobs. In practice, we set $P = 3$ to reconcile the above tradeoff. For search depth D in iterative scaling, increasing D from 1 to 3 raises per-job overhead from 0.88 to 5.98 seconds (asynchronous and non-negligible to LM training), while reducing JCT by 14.6% and improving throughput by 1.03%. We range D between 2 and 5 in practice, with the optimal value depending on workloads and available resources.

Overhead Analysis. We analyze three system overheads: (i) **Job profiling overhead**: With multiple GPUs, the overall complexity is $O(N^2M)/O(MN) = O(N)$, further reducible via skipping repeated operators across grids and limiting grid count to $O(\log N)$. Notably, industrial practice prevents overly large N by assigning dedicated GPUs to hyper-scale jobs rather than dynamic scheduling. The profiling overhead remains <20 minutes (8.5 minutes with $N = 16$ and $M = 4$ in evaluation [35, 42]). (ii) **Job rescheduling overhead** (non-blocking to other jobs) includes space-pruned AP search (1-2 minutes) and checkpoint-resume (<5 minutes [80]), negligible as the average rescheduling count is 2.29 during job lifetime (hours or days). (iii) **Offline profiling overhead** (intra-node communication primitives, one-shot before the cluster comes online) is about 3.5 hours (4-GPU node), negligible to cluster operating lifetime (weeks or months).

Impact on Model Convergence. Arena ensures consistent convergence in dynamic scaling via [35, 53, 79, 82]: (i) fixed hyperparameters (e.g., global batch size) when adjusting DP degrees or GPU resources [35]; (ii) streaming dataset loading that avoids re-training samples after resharding [80]. To illustrate this, we employ Arena to train a GPT-2.6B with *Wikipedia* dataset [34] and shift the DP degree from 1 to 4 (with PP degree of 2). The loss curves exhibit highly similarity with the average mean-square deviation of 0.06.

6 Discussion and Future Work

Intra-Job Heterogeneity. As elaborated in §3.5, Arena scheduler dynamically switches the GPU type for each job while allocating homogeneous GPUs to ensure training efficiency and manageability. With the advent of heterogeneity-aware training frameworks (e.g., Sailor [75], Metis [77]) that parallelize large models on heterogeneous GPUs, Arena remains extensible to integrate them as its training backend. The key modifications include extending the operator load definition and GPU assignment strategy by quantifying the compute capability of different GPU types, and profiling the computation operators for each pipeline stage on a single GPU with the assigned GPU type. We leave the exploration of integrating intra-job heterogeneity as future work.

Extensibility to Training Frameworks. Arena is extensible to integrate other training frameworks (e.g., Megatron-LM [66], PyTorch FSDP [90]) as the AP runtime backend. Taking Megatron-LM as an example, several modifications are necessary since it lacks native support for adaptive parallelism and compilation. First, it is necessary to implement APIs for flexible pipeline stage construction, per-stage GPU assignment, and stage-wise parallelism selection. Second, the framework requires a compiler based on TorchFX [12] to obtain the intermediate representation (IR) graph of Megatron-LM models for operator disaggregation. Lastly, the profiling of computation operators can be performed offline, given that the PyTorch dispatch mechanism guarantees consistent kernel selection for identical input shapes, data types, and hardware [9]. We have developed such a framework as a PyTorch-native training backend, which is open-source together with the JAX implementation of the Arena backend.

7 Related Work

Scheduling Training Jobs. Cluster schedulers for deep-learning (DL) training jobs have surged recently [26, 35, 36, 42, 45, 53, 62, 65, 81, 82, 88, 89]. ElasticFlow [35] elastically scales the number of homogeneous GPUs for in-flight jobs to boost cluster throughput while satisfying job deadlines. Gavel [65] dynamically switches the type of allocated GPUs for jobs to alleviate imbalanced loads across heterogeneous GPU regions. Sia [45] and EasyScale [53] jointly optimize both the number and type of GPUs for more flexible resource allocation. They rely on profiling or estimation for static parallelism (primarily data parallelism), disregarding the performance of adaptive parallelism and thus failing in scheduling large models with complex parallelism strategies.

Parallelizing Large Models. Researchers have widely studied the parallelism optimization of large models [43, 47, 48, 58, 64, 66, 77, 78, 91]. Megatron-LM [66] shards model parameters across multiple GPUs to reduce per-GPU memory footprint. GPipe [43] and PipeDream [64] explore efficient pipeline schedules to enhance overall training throughput. Alpa [91], Unity [78], and Aceso [60] automatically search

the optimal hybrid parallelism strategy on fixed resources. While the training backend of Arena is built on Alpa, its core components are compatible to various training frameworks to provide efficient and precise job performance acquisition for the co-design of scheduling and parallelization.

Modeling Job Performance. Some works [32, 37, 39, 50, 67, 84, 86, 92] have studied modeling the end-to-end performance of DL jobs. NeuSight [50] and Habitat [86] conduct blackbox prediction for kernel-level performance. Centimani [84] and PCS [32] offline profile all operators across multiple GPUs for online modeling. MAD-Max [39] analytically models the latency of communication and computation operators. They focus on zero-GPU prediction or multi-GPU measurement under static parallelism, rather than dynamic (fragmented) resources and adaptive parallelism.

8 Conclusion

We present Arena, a co-designed training system to dynamically schedule and efficiently execute large models with adaptive parallelism in GPU clusters. Arena unifies low-cost profiling and AP-tailored estimation to efficiently navigate the scheduling-parallelism optimization space. Arena conducts dynamic AP-aware scheduling and executes jobs with space-pruned AP. Arena improves cluster throughput by 1.60× and reduces JCT by 49.3% as compared to baselines.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Chuan Wu, for their valuable feedback. This work is partially sponsored by the National Key Research and Development Program of China (2024YFB4505703), National Natural Science Foundation of China (62232011), and Natural Science Foundation of Shanghai Municipality (24ZR1430500). Quan Chen is the corresponding author of this paper.

References

[1] 2024. Amazon EC2 P4 Instance. https://aws.amazon.com/ec2/instance-types/p4/.
[2] 2024. Distributed communication package - torch.distributed. https://pytorch.org/docs/stable/distributed.html.
[3] 2024. Euclidean distance. https://en.wikipedia.org/wiki/Euclidean_distance.
[4] 2024. Hexagonal Sampling. https://en.wikipedia.org/wiki/Hexagonal_sampling.
[5] 2024. JAX: High-Performance Array Computing. https://jax.readthedocs.io/en/latest/index.html.
[6] 2024. Multi-objective optimization. https://en.wikipedia.org/wiki/Multi-objective_optimization.
[7] 2024. Nvidia Mellanox ConnectX-5. https://www.nvidia.com/en-us/networking/ethernet/connectx-5/.
[8] 2024. Nvidia Mellanox ConnectX-6. https://www.nvidia.com/en-sg/networking/ethernet/connectx-6/.
[9] 2024. PyTorch Dispatcher. http://blog.ezyang.com/2020/09/lets-talk-about-the-pytorch-dispatcher/.
[10] 2024. Roofline model. https://en.wikipedia.org/wiki/Roofline_model.
[11] 2024. TensorFlow XLA Compiler. https://www.tensorflow.org/xla.
[12] 2024. Torch FX. https://pytorch.org/docs/stable/fx.html.
[13] 2025. A high performance, open source universal RPC framework. https://grpc.io.
[14] 2025. Kubernetes. https://kubernetes.io/.
[15] 2025. Nvidia Ada Lovelace Architecture. https://www.nvidia.com/en-us/geforce/ada-lovelace-architecture/.
[16] 2025. Nvidia Ampere Architecture. https://www.nvidia.com/en-us/data-center/ampere-architecture/.
[17] 2025. Nvidia Blackwell Architecture. https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/.
[18] 2025. NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl.
[19] 2025. NVIDIA CUDA Profiling Tools Interface (CUPTI). https://developer.nvidia.com/cupti.
[20] 2025. Nvidia Hopper Architecture. https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/.
[21] 2025. Nvidia Volta Architecture. https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/.
[22] 2025. NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/.
[23] 2025. Pareto front. https://en.wikipedia.org/wiki/Pareto_front.
[24] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310* (2024).
[25] 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.
[26] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
[27] 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 178–191. https://doi.org/10.1145/3620666.3651379
[28] Tiancheng Chen, Ales Kubicek, Langwen Huang, and Torsten Hoefler. 2025. CrossPipe: Towards Optimal Pipeline Schedules for Cross-Datacenter Training. *arXiv preprint arXiv:2507.00217* (2025).
[29] 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.
[30] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. arXiv:1604.06174 [cs.LG] https://arxiv.org/abs/1604.06174
[31] Arthur Douillard, Qixuan Feng, Andrei A. Rusu, Rachita Chhaparia, Yani Donchev, Adhiguna Kuncoro, Marc'Aurelio Ranzato, Arthur Szlam, and Jiajun Shen. 2024. DiLoCo: Distributed Low-Communication Training of Language Models. arXiv:2311.08105 [cs.LG] https://arxiv.org/abs/2311.08105
[32] Abdullah Bin Faisal, Noah Martin, Hafiz Mohsin Bashir, Swaminathan Lamelas, and Fahad R. Dogar. 2024. When will my ML Job finish? Toward providing Completion Time Estimates through Predictability-Centric Scheduling. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association,

Santa Clara, CA, 487–505. https://www.usenix.org/conference/osdi24/presentation/bin-faisal

[33] Jared Fernandez, Luca Wehrstedt, Leonid Shamis, Mostafa Elhoushi, Kalyan Saladi, Yonatan Bisk, Emma Strubell, and Jacob Kahn. 2024. Hardware Scaling Trends and Diminishing Returns in Large-Scale Distributed Training. *arXiv preprint arXiv:2411.13055* (2024).

[34] Wikimedia Foundation. 2025. *Wikimedia Downloads*. https://dumps.wikimedia.org

[35] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. ElasticFlow: An Elastic Serverless Training Platform for Distributed Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 266–280.

[36] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning.. In *NSDI*, Vol. 19. 485–500.

[37] 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.

[38] John L Hennessy and David A Patterson. 2017. *Computer architecture: a quantitative approach*. Morgan kaufmann.

[39] Samuel Hsia, Alicia Golden, Bilge Acun, Newsha Ardalani, Zachary DeVito, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. 2023. MAD Max Beyond Single-Node: Enabling Large Machine Learning Model Acceleration on Distributed Systems. arXiv:2310.02784 [cs.DC]

[40] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[41] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. 2024. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 709–729.

[42] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2023. Lucid: A Non-intrusive, Scalable and Interpretable Scheduler for Deep Learning Training Jobs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 457–472.

[43] 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. *Advances in neural information processing systems* 32 (2019).

[44] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. 2023. Tutel: Adaptive mixture-of-experts at scale. *Proceedings of Machine Learning and Systems* 5 (2023), 269–287.

[45] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 642–657.

[46] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads.. In *USENIX Annual Technical Conference*. 947–960.

[47] 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.

[48] Chenyu Jiang, Zhen Jia, Shuai Zheng, Yida Wang, and Chuan Wu. 2024. DynaPipe: Optimizing multi-task training through dynamic pipelines. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 542–559.

[49] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems* 5 (2023), 341–353.

[50] Seonho Lee, Amar Phanishayee, and Divya Mahajan. 2025. Forecasting GPU performance for deep learning training and inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 493–508.

[51] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668* (2020).

[52] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on operating systems design and implementation (OSDI 14)*. 583–598.

[53] Mingzhen Li, Wencong Xiao, Biao Sun, Hanyu Zhao, Hailong Yang, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, et al. 2022. Easyscale: Accuracy-consistent elastic training for deep learning. *arXiv preprint arXiv:2208.14228* (2022).

[54] Qingyuan Li, Bo Zhang, Liang Ye, Yifan Zhang, Wei Wu, Yerui Sun, Lin Ma, and Yuchen Xie. 2024. Flash Communication: Reducing Tensor Parallelization Bottleneck for Fast Large Language Model Inference. *arXiv preprint arXiv:2412.04964* (2024).

[55] Rongzhi Li, Ruogu Du, Zefang Chu, Sida Zhao, Chunlei Han, Zuocheng Shi, Yiwen Shao, Huanle Han, Long Huang, Zherui Liu, and Shufan Liu. 2025. Taming the Chaos: Coordinated Autoscaling for Heterogeneous and Disaggregated LLM Inference. arXiv:2508.19559 [cs.DC] https://arxiv.org/abs/2508.19559

[56] Rui Li, Xiaoyun Zhi, Jinxin Chi, Menghan Yu, Lixin Huang, Jia Zhu, Weilun Zhang, Xing Ma, Wenjia Liu, Zhicheng Zhu, Daowen Luo, Zuquan Song, Xin Yin, Chao Xiang, Shuguang Wang, Wencong Xiao, and Gene Cooperman. 2025. BootSeer: Analyzing and Mitigating Initialization Bottlenecks in Large-Scale LLM Training. arXiv:2507.12619 [cs.LG] https://arxiv.org/abs/2507.12619

[57] 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 *International Conference on Machine Learning*. PMLR, 6543–6552.

[58] Zhiqi Lin, Youshan Miao, Guanbin Xu, Cheng Li, Olli Saarikivi, Saeed Maleki, and Fan Yang. 2024. Tessel: Boosting Distributed Execution of Large DNN Models via Flexible Schedule Search. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 803–816.

[59] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, et al. 2024. {nnScaler}:{Constraint-Guided} Parallelization Plan Generation for Deep Learning Training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 347–363.

[60] 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 *Proceedings of the Nineteenth European Conference on Computer Systems*. 163–181.

[61] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020.

Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 881–897.

[62] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation*.

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

[64] 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.

[65] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 481–498.

[66] 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.

[67] Hang Qi, Evan R Sparks, and Ameet Talwalkar. 2016. Paleo: A performance model for deep neural networks. In *International Conference on Learning Representations*.

[68] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning.. In *OSDI*, Vol. 21. 1–18.

[69] Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. 2023. {AWARE}: Automate workload autoscaling with reinforcement learning in production cloud systems. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 387–402.

[70] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[71] 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. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

[72] Teven Le Scao, Thomas Wang, Daniel Hesslow, Lucile Saulnier, Stas Bekman, M Saiful Bari, Stella Biderman, Hady Elsahar, Niklas Muennighoff, Jason Phang, et al. 2022. What language model to train if you have one million gpu hours? *arXiv preprint arXiv:2210.15424* (2022).

[73] Daniel Snider and Ruofan Liang. 2023. Operator Fusion in XLA: Analysis and Evaluation. *arXiv preprint arXiv:2301.13062* (2023).

[74] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. 2024. ML Training with Cloud GPU Shortages: Is Cross-Region the Answer?. In *Proceedings of the 4th Workshop on Machine Learning and Systems* (Athens, Greece) *(EuroMLSys '24)*. Association for Computing Machinery, New York, NY, USA, 107–116. https://doi.org/10.1145/3642970.3655843

[75] Foteini Strati, Zhendong Zhang, George Manos, Ixeia Sánchez Périz, Qinghao Hu, Tiancheng Chen, Berk Buzcu, Song Han, Pamela Delgado, and Ana Klimovic. 2025. Sailor: Automating Distributed Training over Dynamic, Heterogeneous, and Geo-distributed Clusters. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 204–220. https://doi.org/10.1145/3731569.3764839

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

[77] Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, Mohd Muzzammil, and Myeongjae Jeon. 2024. Metis: Fast Automatic Distributed Training on Heterogeneous {GPUs}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 563–578.

[78] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. 2022. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 267–284.

[79] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. 2024. Tenplex: Dynamic Parallelism for Deep Learning using Parallelizable Tensor Collections. https://doi.org/10.1145/3694715.3695975 arXiv:2312.05181 [cs.DC]

[80] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, et al. 2024. ByteCheckpoint: A Unified Checkpointing System for Large Foundation Model Development. *arXiv preprint arXiv:2407.20143* (2024).

[81] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 22)*.

[82] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 595–610.

[83] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning.. In *OSDI*. 533–548.

[84] Zhen Xie, Murali Emani, Xiaodong Yu, Dingwen Tao, Xin He, Pengfei Su, Keren Zhou, and Venkatram Vishwanath. 2024. Centimani: Enabling Fast AI Accelerator Selection for DNN Training with a Novel Performance Predictor. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 1203–1221. https://www.usenix.org/conference/atc24/presentation/xie

[85] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. 2023. SkyPilot: An Intercloud Broker for Sky Computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 437–455. https://www.usenix.org/conference/nsdi23/presentation/yang-zongheng

[86] Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. 2021. Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 503–521.

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

[88] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Mingxia Li, Fan Yang, Qianxi Zhang, Binyang Li, Yuqing Yang, Lili Qiu, Lintao Zhang, and Lidong Zhou. 2023. SiloD: A Co-design of Caching and Scheduling for Deep Learning Clusters. In *Proceedings of the Eighteenth*

European Conference on Computer Systems (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 883–898. https://doi.org/10.1145/3552326.3567499

[89] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis CM Lau, Yuqi Wang, Yifan Xiong, et al. 2020. HiveD: Sharing a GPU cluster for deep learning with guarantees. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 515–532.

[90] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. arXiv:2304.11277 [cs.DC] https://arxiv.org/abs/2304.11277

[91] 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.

[92] 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)*. USENIX Association, 337–352.

A Artifact Appendix

A.1 Abstract

Arena is a large model training system to dynamically schedule and efficiently execute large models with adaptive parallelism in GPU clusters. This artifact includes the source codes of the Arena prototype and the benchmarking instructions to evaluate its functionality and reproduce its major experimental results.

A.2 Description & Requirements

A.2.1 How to access. The Arena system is available at: GitHub https://github.com/sjtu-epcc/arena/tree/ae-eurosys# and Zenodo https://zenodo.org/records/18870526. More detailed instructions are provided in the repository `README`.

A.2.2 Hardware dependencies. The artifact requires a Linux system equipped with at least 192 GB of system memory, 256 GB of available disk storage, and 4 NVIDIA A40 GPUs (48GB, connected via PCIe or NVLink). Notably, given that the full-fleet evaluation involves tens to hundreds of GPUs, for reproducibility, the artifact mainly uses 4 NVIDIA A40 GPUs unless specified.

A.2.3 Software dependencies. The artifact requires Conda for package management. The software stack includes CUDA 11.8 and Python3.8. All software dependencies are automatically installed in our provided Docker image.

A.3 Set-up.

Users should clone the repository of Arena system and build from the Docker image.

```
# Git clone
git clone --recursive -b ae-eurosys https://github.
    com/sjtu-epcc/arena.git
cd arena
git checkout ae-eurosys
# Docker build, run, and backend installation
cd runtime
docker build -t arena/arena:ae-eurosys -f ./
    profile_cu118.Dockerfile .
docker run --runtime=nvidia -it --rm --gpus all --shm-
    size 64g --network=host --privileged --volume [
    USER_DIR]/.cache:/root/.cache --env
    NVIDIA_DISABLE_REQUIRE=1 --name arena arena/arena
    :ae-eurosys
conda activate alpa       # Alpa env
bash jaxpr/cpp/install.sh # Build cpp backend
```

A.4 Evaluation Workflow

A.4.1 Major claims. The major claims of Arena system for artifact evaluation include:

- **(C#1)** The disaggregated profiler of Arena achieves average error rates of 4.4%, 5.1%, 3.1%, 4.6%, and 8.3% for 1, 2, 4, 8, and 16 GPU cases; Arena reduces the GPU time (i.e., elapsed time × occupied GPU count) by 18.1× on average (2.55× at least), as compared to direct measurement.
- **(C#2)** In the parallelism planner of Arena, the best proxy plan (used for scheduling) among grids achieves average 93.4% performance of the AP searched optimal plan, thus is accurate enough to achieve AP-aware cluster scheduling.
- **(C#3)** With AP-aware scheduling, Arena scheduler reduces average job completion time (JCT) by 81.3% (FCFS), 80.5% (ElasticFlow-LS), 76.6% (Gavel) and 75.2% (Sia), completing up to 1.45× more jobs. From the cluster perspective, Arena outperforms baselines with up to 1.55× higher average throughput and 1.58× higher peak throughput.

A.4.2 Disaggregated profiling experiment (C#1). We provide the instructions to run the single-device profiler and the multi-device direct execution to evaluate the accuracy and profiling cost reduction.

Before running single-device profiling, users should offline profile the communication latency data (may take dozens of minutes or a few hours). Detailed instructions are provided in repository `README`.

Users can evaluate single-device profiling by specifying (1) model configurations (layers are uniformly clustered into stages), (2) device assignments, and (3) parallelism plans. Taking vanilla 1F1B pipeline parallelism on 4 GPUs as an example (**full instructions in repository `README`**, "Crius" is the alias for "Arena", "cell" for "grid"):

```
export ENABLE_CRIUS_PROFILER=true # Enable arena
export CUDA_VISIBLE_DEVICES=0 # Specify a single GPU
python jaxpr/runtime_profiler.py --estimate_e2e --
    num_hosts 1 --num_devices_per_host 4 --
    devices_name 1_a40 --model_name wide_resnet --
    param_num 1B --batch_size 256 --parallel_degrees
    =4,1,1 # [pp,dp,tp]
```

To measure the end-to-end iteration time (rather than estimating it) with the specified parallel plan:

```
export CUDA_VISIBLE_DEVICES=0,1,2,3 # Specify all
ray start --head  # Start ray cluster
export ENABLE_CRIUS_PROFILER=false # Disable arena
python jaxpr/runtime_profiler.py --measure_with_alpa
    --num_hosts 1 --num_devices_per_host 4 --
    devices_name 1_a40 --model_name wide_resnet --
    param_num 1B --batch_size 256 --parallel_degrees
    =4,1,1 # [pp,dp,tp]
```

The evaluation results are output in the console logs. In our 1×4 A40 node, the estimated/measured end-to-end iteration time is 15.893s/16.121s with the profiling cost of 63.475/613.93 #GPU × time(s).

A.4.3 Parallelism planning experiment (C#2). We then provide the instructions to run the parallelism planner of Arena to evaluate the performance of the best proxy plan (used for cluster scheduling) among all grids, compared to the AP searched optimal plan from Alpa.

To evaluate the parallelism planner among all grids, i.e., the best proxy plan (used for scheduling this job) among all grids:

```
export ENABLE_CRIUS_PROFILER=true # Enable arena
export CUDA_VISIBLE_DEVICES=0 # Specify one GPU
python jaxpr/crius_cell_profile.py --num_hosts 1 --
    num_devices_per_host 4 --devices_name 1_a40 --
    model_name wide_resnet --param_num 1B --
    batch_size 256 --num_micro_batches 16 --
    cell_prof_strategy=auto
```

To measure the performance of the optimal parallelism plan searched by Alpa:

```
export CUDA_VISIBLE_DEVICES=0,1,2,3 # Specify all
ray start --head  # Start ray cluster
export ENABLE_CRIUS_PROFILER=false # Disable arena
python jaxpr/runtime_profiler.py --optimize_with_alpa
     --num_hosts 1 --num_devices_per_host 4 --
    devices_name 1_a40 --model_name wide_resnet --
    param_num 1B --batch_size 256
```

The evaluation results are output in the console logs. In our 1×4 A40 node, the end-to-end iteration times of Arena estimated and Alpa optimized are 10.445s and 10.633s, respectively.

A.4.4 Large-scale simulated scheduling experiment (C#3). The simulation requires offline profiling all training jobs by enumerating possible combinations of models (e.g., GPT-1.3B), hyperparameters (e.g., global batch size 256), and allocated hardware (e.g., 1×4 A40 GPUs). Here, to avoid extensive offline profiling for artifact evaluation, we have provided our profiling data in `./database/prof_database.pkl` that includes Arena's estimated data, data profiled via data parallelism, and Alpa's searched data. Notably, **running this experiment requires no GPU resources**, and users can either copy the scheduler-related codes into the container via `docker cp ./ arena:/app/` or directly execute the simulator on the host (need to manually install minor related dependencies).

To run large-scale simulated scheduling with 1,280 GPUs and Philly trace, use the following instructions (`[POLICY]` includes `fcfs`, `elasticflow-l`, `gavel`, and `sia`):

```
# For Arena
python simulator.py --policy=crius --trace_type=
    philly --sched_with_opt --max_sched_round=2000 --
    enable_alpa --result_dir=./plot
# For other baselines
python simulator.py --policy=[POLICY] --trace_type=
    philly --max_sched_round=2000 --enable_alpa --
    result_dir=./plot
```

We also provide scripts to visualize the results (`[METRIC]` includes `thr`, `jct`, and `queuing_time`):

```
python simulator.py --visual --visualized_metric=[
    METRIC] --result_dir=./plot --out_dir=./figures
    --trace_type=philly
```

For throughput metric, users can both inspect average/-maximum values in console logs and visualized curves (Figure 11) in `./figures/cluster_thr.pdf`. For JCT, number of finished jobs, and queuing time metrics, users can inspect results in console logs.