Suika: Efficient and High-quality Rescheduling of 3D-parallelized LLM Training Jobs in Shared Clusters

Yuxuan Wang*
fAKe@sjtu.edu.cn
Zhiyuan College, Shanghai Jiao Tong University
Institute of Artificial Intelligence (TeleAI), China Telecom

Yanbo Wang
wang-yanbo@sjtu.edu.cn
Shanghai Jiao Tong University
Institute of Artificial Intelligence (TeleAI), China Telecom

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

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

Qizhen Weng
wengqzh@chinatelecom.cn
Institute of Artificial Intelligence (TeleAI), China Telecom

Yin Chen
cheny304@chinatelecom.cn
Institute of Artificial Intelligence (TeleAI), China Telecom

Zeren Li
lizeren2@huawei.com
Huawei Technologies Co., Ltd.

Xuqi Zhu
zhuxuqi@huawei.com
Huawei Technologies Co., Ltd.

Yongqiang Yang
yangyongqiang@huawei.com
Huawei Technologies Co., Ltd.

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

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

Abstract

Large Language Models (LLMs) are usually trained with 3D (data, tensor, and pipeline) parallelism—in shared GPU clusters where the available resources are highly dynamic. Rescheduling the idle resources to ongoing jobs can help improve cluster utilization, but doing so for 3D-parallelized training jobs suffers large overheads in performance modeling, decision making, and redeployment. We present *Suika*, a cluster training system that supports efficient and high-quality resource rescheduling for 3D-parallelized LLM training jobs. *Suika* holistically addresses the complexity challenges by exploiting the *incremental* nature of rescheduling. For performance modeling, it builds an accurate performance estimator with non-disruptive online profiling. For decision-making, it employs topology-aware sorting and an expand-and-balance algorithm to reduce the complexity of resource allocation and job parallelization, without compromising decision quality. *Suika* further integrates a device-to-device redeployment method to leverage the overlapping nature of incremental reconfiguration for overhead reduction. Experiments on 64-GPU physical cluster and 1024-GPU simulated cluster show that, *Suika* achieves 1.29 ∼ 1.31× reduction in average JCT compared to state-of-the-art schedulers.

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

Keywords: Job Scheduling, LLM Training, Resource Efficiency

ACM Reference Format:
Yuxuan Wang, Yanbo Wang, Chen Chen, Chunyu Xue, Qizhen Weng, Yin Chen, Zeren Li, Xuqi Zhu, Yongqiang Yang, Quan Chen, and Minyi Guo. 2026. *Suika*: Efficient and High-quality Rescheduling of 3D-parallelized LLM Training Jobs in Shared Clusters . In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3767295.3803623

*Part of this work was conducted during an internship at TeleAI.
†Chen Chen is the corresponding author.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
EUROSYS '26, Edinburgh, Scotland Uk
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2212-7/2026/04
https://doi.org/10.1145/3767295.3803623

1 Introduction

Large Language Models (LLMs) [3, 54, 61–63], which comprise tens of Transformer-based neural network layers, have been widely adopted in modern society as a pioneering AI technique [66]. Due to the compute- and memory-intensive nature of LLM training, LLM training jobs are typically conducted in shared GPU clusters [19], with each job parallelized over multiple GPUs. Data Parallel (DP) [8, 34], Pipeline Parallel (PP) [21, 44] and Tensor Parallel (TP) [45, 57] are the most representative parallelization paradigms, often adopted jointly under a 3D-parallelized deployment plan [56, 57, 70].

Although additional GPU resources may become available at arbitrary moments in a shared GPU cluster, a 3D-parallelized job, once deployed, nonetheless cannot flexibly scale out to exploit these temporarily idle GPUs in practice. This results in resource wastage and prolonged job completion time.

Therefore, we aim to enable practical resource rescheduling for 3D-parallelized LLM training jobs. Upon the release of GPUs from a completed job, the resources should be reallocated to ongoing jobs that yield the highest marginal performance improvement. Given the high degree of dynamicity in shared clusters, such rescheduling must be performed with minimal overhead.

However, achieving efficient and high-quality resource rescheduling is challenging for 3D-parallelized jobs. In essence, resource rescheduling has three key steps: (1) build performance models for each job to predict the throughput under arbitrary 3D-parallelization plans, (2) decide the optimal allocation of the idle resources among the ongoing jobs with their parallelization plans accordingly adapted, and (3) redeploy the affected jobs to switch to the new parallelization plans. Each step incurs large overheads: for performance modeling, existing methods [64, 79, 81] require a series of disruptive profiling runs; for decision-making, both resource allocation and parallelization plan have an exponential solution space, which existing methods [25, 53, 79] fail to prune without quality degradation; for redeployment, the standard checkpoint-resume method [38] incurs large communication overhead and initialization latency. A practical rescheduling system must attain time-efficiency in all these aspects while maintaining high decision quality.

To that end, our insight is that job rescheduling—where the candidate jobs to receive additional resources are already running—can be conducted in an *incremental* manner. The benefits are *three-fold*. First, the job's current training process is a *costless angle for bootstrapping performance model*: by analyzing rich information of the job's current execution status, it is possible to extrapolate its performance with unseen parallelization plans. Second, the current training process presents an *effective shortcut for making rescheduling decisions*: the optimal parallelization plan to a few additional GPUs is usually similar with the original one, which is helpful to prune the solution space. Third, the current training process also provides a *reusable starting-point for mitigating redeployment overheads*: when redeploying a job over expanded resources, it is likely that the intermediate training states on a previously-adopted GPU can be reused.

In this paper, we propose *Suika*, a holistic system that supports efficient and high-quality rescheduling for 3D-parallelized LLM training jobs. *Suika* comprises three modules: *SuikaAgent*—conducting non-disruptive performance modeling and parallelization planning for one job, *SuikaSched*—promptly allocating cluster resources among competing jobs for the best overall performance, and *SuikaEngine*—enforcing fast redeployment with minimum delay in the critical path. We next respectively elaborate the innovations in each module.

First, in *SuikaAgent*, we adopt *full-spectrum profiling* to model the throughput of a 3D-parallelized job without interruptive profiling runs. We unify existing performance models [64, 79, 81] under a common dependency graph, which reveals that the number of unknown coefficients is less than the number of observable performance aspects—a perspective existing methods ignored. That is, we can model the performance of a 3D-parallelized job by merely profiling its runtime execution status—in *one shot* but from *multiple aspects*, instead of in multi-shots (with multiple trial runs) but each time from a single aspect. Evaluation reveals that *Suika*, with zero extra profiling overhead, successfully achieves similar accuracy as disruptive methods.

Meanwhile, in *SuikaSched*, we devise multiple techniques to make efficient and high-quality scheduling decisions. When making rescheduling decisions there are two key complexity challenges: (1) what GPU subset to allocate for each job, and (2) what parallelization plan to use for a job when a GPU subset is added—both with an exponential solving complexity. To reduce the allocation complexity, for each job we enforce a rigid GPU provisioning order based on the topology affinity; to reduce the parallelization complexity, we derive new parallelization plans in an *expand-and-balance* manner with scalable dynamic programming. Moreover, for inter-job scheduling, we propose to conduct multi-round allocation with controlled expansion aggressiveness, so as to maximize the overall resource rescheduling effect without hurting future-arrival jobs.

Additionally, in *SuikaEngine*, we fully exploit the incremental nature for lightweight redeployment. To reduce the redeployment latency, we overlap runtime initialization with the original training process and enable device-to-device transfer to avoid remote checkpointing. Apart from these basic techniques (shared by some recent works [31, 67, 72]), *SuikaEngine* also makes special customizations to further enhance the redeployment efficiency. For a common case where a model layer is hosted on the same GPU as before, we adopt the CUDA IPC (inter-process communication) technique [49] to bypass GPU-CPU transmission; To handle extra memory overhead with in-GPU migration, we adopt *chunked transfer* to avoid OOM (out-of-memory) failure during IPC. With these optimizations combined, *Suika* attains 7× faster redeployment over naive checkpoint-resume method.

We have implemented our *Suika* system in around 9K LoC, and have conducted both testbed and simulation experiments to evaluate its performance superiority. In a 64-GPU physical cluster training diverse LLM models, *Suika* can improve the cluster utilization by 59.5%, with an average JCT reduction of 1.67×. Meanwhile, compared with state-of-the-art elastic schedulers like *Sia* [25] and *Rubick* [79], *Suika* achieves a JCT improvement of over 1.29×—with its fast-yet-high-quality rescheduling and lightweight redeployment.

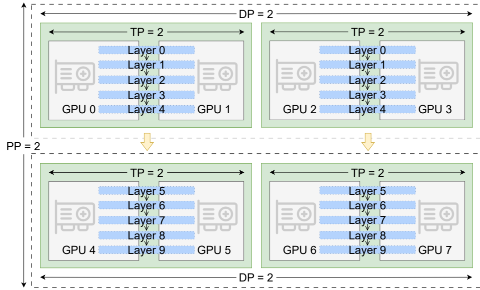


Figure 1. A typical 3D-parallelized deployment of a 10-layer LLM over 8 GPUs, following the PP-DP-TP hierarchy.

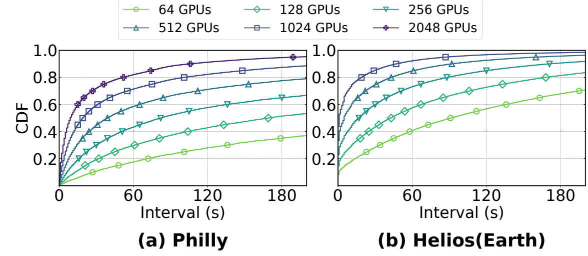


Figure 2. CDFs of time intervals between two consecutive job events (down-sampled to emulate different cluster scales).

We also checked the effectiveness and robustness of each *Suika* module in diverse setups, and our simulation further confirms that *Suika* scales well to large clusters with over one thousand GPUs.

2 Background and Motivation

2.1 3D-parallelized LLM Training in Shared Clusters

LLM basics. Large Language Models (LLMs) have substantially revolutionized diverse fields such as text processing [1, 13, 78], code generation [7, 37, 47] and embodied AI [40, 59, 60]. LLMs employ a Transformer-based architecture consisting of dozens to hundreds of identical layers [66]. A prerequisite for applying LLMs is the training of powerful models with large-scale datasets (including both pre-training [10, 77] and fine-tuning [9, 17, 75]); such LLM training jobs have become popular AI workloads in production GPU clusters [19].

3D-parallelism in LLM training. Due to massive parameter size and training data volume, as depicted in Fig. 1, an LLM is often parallelized across multiple GPUs using data, tensor, and pipeline parallelism [19, 45, 56, 57, 70]. *Data parallelism* (DP) [8, 34] replicates model parameters on parallel devices, each processing disjoint data subsets and synchronizing gradients before updates. *Tensor parallelism* (TP) [45, 57] partitions intra-operator computation across devices, reducing per-device memory usage but incurring high communication overhead. *Pipeline parallelism* (PP) [21, 44] splits the model into multiple stages and processes data in microbatches, with relatively low communication overhead.

Effective composition of these techniques (termed *3D-parallelism*) is the key to LLM training performance. Popular systems such as *Megatron-LM* [45] and *DeepSpeed* [56] allow users to manually design uniformly-partitioned 3D-parallelization plans over symmetric resources. Subsequently, more advanced algorithms like *Alpa* [81] and *Metis* [64] have been proposed to automatically generate competitive 3D-parallelization plans for general scenarios even with heterogeneous resources, typically following a PP-DP-TP hierarchy shown in Fig. 1.

Resource rescheduling demand in shared clusters. Due to the high infrastructure cost, LLM training jobs from different tenants often share a GPU cluster [18, 26]. In such shared clusters, jobs may arrive and complete at arbitrary moments (which we call *job events*), rendering the available resources highly dynamic. To elaborate, Fig. 2 depicts the interval distribution between two consecutive job events in two public cluster traces (down-sampled to different scales)—Philly [26] from Microsoft, and Helios [18] from SenseTime. Fig. 2 reveals that job events are increasingly common in larger clusters: for example, for a cluster with 1024 GPUs, 70% of such intervals are shorter than 60s. In that sense, it is highly likely that, after an LLM training job starts, additional GPU resources may become available to it.

To harvest such temporally available GPUs to accelerate ongoing 3D-parallel jobs, we need to make timely resource rescheduling, which involves allocating newly-released GPUs to the most suitable job and also adapting its parallelization plans. In particular, to make such rescheduling truly beneficial to the end-to-end job performance, it must be conducted (1) *time-efficiently*—so that the overall rescheduling overhead is low despite the frequent resource fluctuations, and (2) *also with high quality*—so that the training speedup under the new 3D-parallel plan is salient.

2.2 Challenges for Rescheduling 3D-parallelized Jobs

With rescheduling enabled, when new GPU resources are released, the scheduler must determine (1) which running jobs to allocate these resources and (2) how to adjust their parallelization plans for the best performance.

Formally, let R^{Δ} denote the newly released GPU resources and $\mathcal{J}$ the set of running jobs, where each $J \in \mathcal{J}$ has a current allocation R_J^0 and receives $R_J^{\delta} \subseteq R^{\Delta}$ additional resources. For each job J and allocation R_J, let $\mathcal{P}(J, R_J)$ be the set of valid parallelization plans; let $\mathrm{T}(P_J)$ be the throughput for $P_J \in \mathcal{P}(J, R_J)$, and $\bar{\mathrm{T}}(J, R_J)$ be the throughput with optimal parallelization plan under allocation R_J. Since absolute throughputs across jobs are not comparable, we measure the the normalized improvement $\bar{\mathrm{T}}_{\text{norm}}$, and weight[1] each job

[1]Such a weighted form of $\bar{\mathrm{T}}_{\text{norm}}$ is reasonable because, from the resource utilization perspective, we shall not prefer "accelerating 10 single-GPU jobs

by the size of current allocation $\|R_J^0\|$ (i.e., GPU count; or sum of TFLOPs if multiple GPU types co-exist). The resulting optimization problem is

$$\begin{aligned} \max_{\{R_J^\delta\}_{J\in\mathcal{J}}} \quad & \sum_{J\in\mathcal{J}} \|R_J^0\| \cdot \bar{\mathrm{T}}_{\mathrm{norm}}(J, R_J^0, R_J^\delta), \qquad (1) \\ \text{s.t.} \quad & \bigsqcup_{J\in\mathcal{J}} R_J^\delta \subseteq R^\Delta, ^2 \\ \text{where} \quad & \bar{\mathrm{T}}(J, R_J) = \max_{P_J\in\mathcal{P}(J,R_J)} \mathrm{T}(P_J), \\ & \bar{\mathrm{T}}_{\mathrm{norm}}(J, R_J^0, R_J^\delta) = \frac{\bar{\mathrm{T}}(J, R_J^0 \cup R_J^\delta)}{\bar{\mathrm{T}}(J, R_J^0)}. \end{aligned}$$

Solving the problem takes three steps: (1) Prior to decision-making, thoroughly profile the training job to establish estimation of $\mathrm{T}(P_J)$ for all possible P_J. (2) During decision-making, enumerate vast partitions of R^Δ to R_J^δ, respectively find the optimal plan for each J, and estimate the overall efficiency. (3) After decision-making, checkpoint-resume the selected jobs to switch to new configurations. However, this naive method incurs impractical overheads in all three steps.

Challenge I: Disruptive profiling procedures. Informed scheduling requires accurate estimation of $\mathrm{T}(P_J)$, but exhaustively profiling every plan in the combinatorial space $\mathcal{P}(J, R_J)$ is infeasible. Prior works instead use analytical performance models calibrated with profiling data. Solvers like *Alpa* [81] and *Metis* [64] profile per-layer latencies under various parallelization strategies. *Rubick* [79], an elastic scheduler that also models 3D-parallelism, builds a fully-parametric model that requires fitting job-specific coefficients with the execution data from multiple trial runs.

A key drawback is that complex profiling procedure must be carried before training commences, which burdens users, consumes valuable cluster resources, and delays job execution, rendering it *disruptive and undesirable*. For example, *Metis* reports 12.5 to 18.9 minutes of profiling overhead for $4 \sim 64$-GPU job, and *Rubick* requires 7 profiling runs across different configurations for each job.

Challenge II: High decision-making complexity. Problem 1 contains two `max` operators respectively at intra- and inter-job level, both bringing substantial solution complexity.

First, at the intra-job level (i.e., for $\max_{P_J\in\mathcal{P}(J,R_J)} \mathrm{T}(P_J)$), the scheduler must search for the optimal parallelization plan after a job receives additional resources. This by default requires exploring a number of candidate plans that grows exponentially with GPU count. Even after pruning, state-of-the-art solvers' searching overheads remain prohibitive for online decision-making: on a 64-GPU setup, both *Alpa* [81] and *Metis* [64] require over 600s to generate a single plan.

by 10%" over "accelerating 1 ten-GPU job by 10%". This way, we can depict the true *utility-per-TFLOP* of the additional resources.

[2] $\bigsqcup$ denotes non-intersect union.

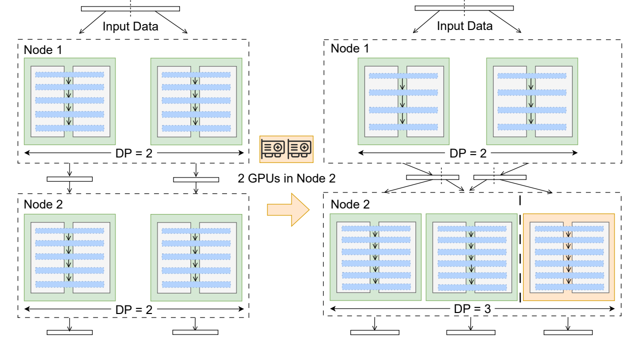


Figure 3. For an LLM spanning over 2 nodes (each with 4 GPUs in 2 DP groups), when added 2 GPUs from Node-2, the best parallelization plan becomes asymmetric (with the layers proportionally allocated to the two nodes).

To reduce such intra-job solution complexity, existing elastic schedulers degrade $\mathcal{P}(J, R_J)$ to either *1D-Parallelism (DP-only)* [2, 23, 25, 53] or *Symmetric 3D-Parallelism* [79]. Schedulers in the former category assume that no TP/PP scheme is adopted (like in *Pollux* [53]) or only expand DP with the TP/PP configuration fixed (like in *Sia* [25]). For the latter category, *Rubick* [79] is a representative scheduler. It maintains a static resource sensitive curve that maps the number of GPUs to its best parallelization plan, which is obtained by enumerating all *symmetric* 3D-parallelism plans (i.e., resources equally divided in each dimension).

However, both simplification methodologies compromise the solution quality. Consider the example in Fig. 3: the job is running with ($PP = 2, DP = 2, TP = 2$), where the first stage is placed on node-1 and the second on node-2; now 2 more GPUs from node-2 become available. In that case, an ideal parallelization plan is to add the idle GPUs to stage-2 as a new DP group with TP=2, which speeds up the job by 22% (measured for a LlaMA-7B job in our GPU testbed described in §5.1). Yet, existing studies fail to find such an asymmetric plan due to their over-pruned $\mathcal{P}(J, R_J)$ solution space.

Second, at the inter-job level, the scheduler must determine a partition of R^Δ among multiple jobs so that the overall efficiency is maximized, which also involves exponential possibilities. Beyond the rescheduling scenario, the scheduler needs to strike a delicate balance between optimistically accelerating ongoing jobs and provisioning resources for future jobs—which must not be starved and may exhibit better scaling efficiency.

To summarize, no method so far can solve Problem 1 in an efficient manner while guaranteeing high solution quality.

Challenge III: Non-negligible redeployment overheads. When a reconfiguration occurs, the LLM training job needs to be redeployed following the new parallelization plan. In

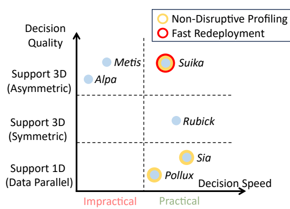


Figure 4. *Suika* superiority over core related works.

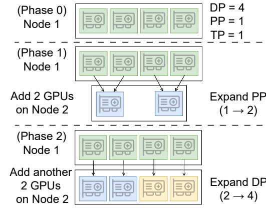


Figure 5. Incremental resource expansion in 2 steps.

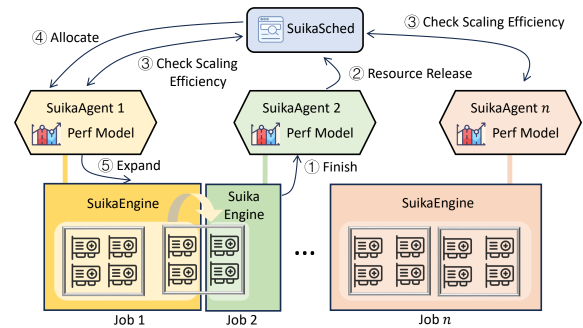


Figure 6. Architecture and workflow of *Suika*.

existing schedulers [25, 53, 79], this process requires checkpointing the full training state to remote storage and reloading it onto the assigned GPUs, which incurs substantial communication overhead. For example, a prior work [38] reports 1.38 and 2.02 minutes spent on checkpoint-resume with 7B and 13B models. As model sizes grow, these costs increase accordingly. Meanwhile, a report [19] reveals a median job duration of only 2 minutes in two LLM deployment clusters, and emphasizes the recent trend towards a shorter job duration distribution. To fully utilize the elastic opportunity under frequent resource fluctuations (as depicted in Fig. 2), the redeployment cost would no longer be negligible.

Design objective. Given the above analysis, as shown in Fig. 4, our objective is thus to create a practical scheduling system that can simultaneously attain high decision quality (as near-optimal as *Metis*) and high efficiency (comparable to *Rubick* and *Sia* in decision-making speed, yet with much faster profiling and redeployment).

2.3 Insight: Exploiting the Incremental Nature

For efficiency and high quality, we find that job rescheduling should be more effectively conducted in an *incremental* manner (instead of conducted from scratch). Specifically, such *incremental reconfiguration* presents three opportunities:

(1) Non-disruptive performance modeling via online profiling. At a rescheduling event, candidate jobs to receive additional resources are already running. A job's current execution state contains abundant information regarding computation, communication, and GPU memory usage patterns. When new resources become available, these detailed measurements can be extrapolated to accurately predict the performance of new parallelization plans, eliminating the need for disruptive profiling. For the example in Fig. 3, the computation latency per layer can be reused, and the data-parallel synchronization time can be estimated analytically as volume/bandwidth. In this way, we predict the end-to-end latency of new parallelization plan with only 1.9% error.

(2) Efficient decision-making via incremental parallelization. High-throughput parallelization plans share common features—balanced computation and communication, similarity in structure, etc. When the current plan is well-balanced, incorporating a small set of additional resources may not necessitate a global re-optimization. Instead, we may derive new plans with *incremental* adjustments along the DP, TP, and PP dimensions. For example in Fig. 3, the new plan can be considered as adding 2 GPUs to stage2 in DP dimension, and moving 1 layer from stage1 to stage2 to rebalance the workload. Similarly, in Fig. 5, we add 4 GPUs in two phases, where phase1 expands in PP dimension, and phase2 expands in DP dimension; *Metis* [64] respectively explores 41 and 135 plans for phase1/2 and eventually lands the same plans as incremental adjustment, which examines fewer than 10 candidate plans. In that sense, the current 3D-parallelization plan can help promptly determine whether to take the candidate resource and how to adapt the parallelization plan.

(3) Fast redeployment via minimal device-to-device migration. With incremental parallelization, both the new resource allocation and parallelization plan maintain a large overlap with the previous ones. Consequently, most of the model and optimizer state already reside on the correct devices. This property allows redeployment to be a minimal device-to-device transfer over the high-bandwidth training network, instead of a checkpoint-resume of the entire state. This method bypasses network bottleneck at remote storage [71]. For the example in Fig. 3, redeployment only requires transferring 1 layer from stage1 to stage2 and populating the new workers with 5 layers, reducing the data movement to only 13% compared to checkpoint-resume.

To summarize, by exploiting the incremental nature of job rescheduling, we can achieve lightweight yet high-quality performance modeling, decision making, and job redeployment. In the next section, we show how such an insight can be implemented in practice.

3 *Suika* Design

In this paper, we design *Suika*, a holistic system that supports efficient and high-quality rescheduling of 3D-parallelized jobs. As shown in Fig. 6, *Suika* has three key components: (1) *SuikaSched*—which dynamically allocates GPUs among jobs; (2) *SuikaAgent*—each manages the performance model

and parallelization plan for one job; (3) *SuikaEngine*—the training runtime that executes a 3D-parallelized job with fast-redeployment support.

Fig. 6 also depicts the *Suika* workflow when a rescheduling event occurs. Specifically, when ① one job (e.g., Job-2 in Fig. 6) completes, its *SuikaEngine* would first signal its *SuikaAgent*, which subsequently ② notifies the *SuikaSched* of the GPUs released. ③ In response, the *SuikaSched* communicates with the *SuikaAgent* of each ongoing job to find the best resource allocation target(s) (Job-1 in Fig. 6). ④ It would notify the identified *SuikaAgent* on the additionally allocated GPUs, which finally ⑤ instructs the *SuikaEngine* on the new parallelization plan to deploy.

Next, we detail the key designs of each *Suika* component.

3.1 Non-Disruptive Performance Modeling

Accurate performance prediction for arbitrary parallelization plans is critical for making high-quality scheduling decisions, which is a classical research problem in the literature. Suika adopts standard modeling assumptions—including linear relationships, bandwidth constraints, and computation-communication overlap—that have proven both generalizable and effective in established systems [31, 64, 79, 81]. Building on these prior works, Suika constructs a generic performance model that supports asymmetric and heterogeneous configurations via a top-down PP-DP-TP hierarchy.

Fig. 7 illustrates[3] the dependency graph to estimate iteration latency T_{PP}. At the top level, T_{PP} breaks into three time periods: $T_{warm\text{-}up}$ for the first micro-batch; T_{steady} for remaining micro-batches; T_{extra} for DP synchronization and optimizer update time that cannot be fully overlapped by computation. All these values can be calculated with $T_{DP\text{-}comp}$, $T_{DP\text{-}comm}$, $T_{DP\text{-}optim}$ (i.e., the computation, synchronization, and optimizer time of each stage); each is further decomposed to lower-level factors. Specifically, the root (lowest-level) vertices in the dependency graph are free coefficients: some—like k_{param} (parameter size)—are *white-box coefficients* known a priori, yet some others—like $k_{overlap}$ (computation-communication overlap ratio) and k_{comp} (net computation constant)—are *hidden coefficients* that must be acquired by profiling. With a performance model, for a given job and its parallelization plan, to make performance prediction is to calculate the top-level T_{PP} with the lower-level factors.

Essentially, to establish the performance model is to calibrate hidden coefficients with profiled observations. Mathematically, n free coefficients require at least n observations. However, existing methods measure *only one factor per configuration*—*Rubick* measures T_{PP} only: there are 7 hidden coefficients in its model, which are then fitted with 7 end-to-end latencies (T_{PP}) via regression. *Alpa* and *Metis* calculate T_{PP} by directly profiling $T_{DP\text{-}comp}$: for each setup under

[3] *Suika* also features a GPU memory prediction model, with an average error of only 3%, but is not included in the figure.

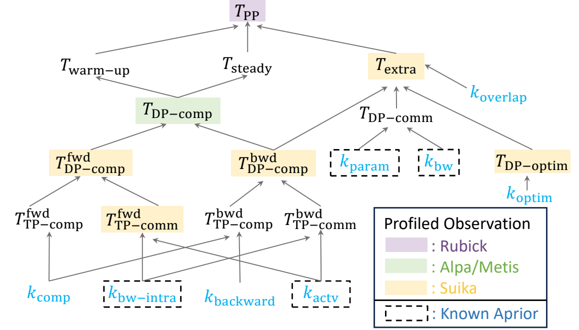


Figure 7. Dependency graph to estimate iteration time T_{PP} (the derivation process is detailed in Appendix A).

$T_{DP\text{-}comp}$ they launch a dedicated trial run, with all the lower-level factors bypassed. In theory, given the relatively large degree-of-freedom to solve, these approaches always require disruptive profiling over multiple configurations.

Suika addresses this inefficiency with a technique we call *full-spectrum profiling*—by noticing that the running status under one configuration suffices for retrieving all the hidden coefficients. As shown in Fig. 7, there are 4 white-box coefficients and 5 profile-able performance metrics (marked by yellow shadows), and only 4 coefficients remain hidden. By leveraging the graph dependencies, *Suika* is able to resolve all hidden coefficients (each dependency level can be inverted into an equation, most of which are linear). Due to the space limitation, we place the detailed mathematical formulation and steps to solve all hidden coefficients in Appendix A. With full-spectrum profiling, *Suika* can accurately infer all the coefficients merely from the ongoing job's runtime status, without the need for any offline profiling.

Evaluation reveals that *Suika* achieves similar or even slightly better accuracy over disruptive methods. Beyond accuracy, one might naturally question the stability of online profiling. We observe that transient instability appears only during initial warm-up; Suika delays reconfiguration until coefficients stabilize (typically within tens of seconds), after which the coefficients remains robust with $\leq 2\%$ fluctuation.

3.2 Efficient and High-quality Planning

In this part, we show how *Suika* exploits the incremental nature to accomplish time-efficient yet high-quality decision-making. Recall that in Problem. 1, there are two complexity challenges: (1) the vast number of candidate (J, R^{δ}_{J}) pairs where $R^{\delta}_{J} \subseteq R^{\Delta}$; (2) the vast number of candidate parallelization plans for each (J, R^{δ}_{J}) pair. *Suika* substantially reduces the overheads in both aspects while preserving high scheduling quality. Besides, it also ensures that the expansion of ongoing jobs does not negatively affect future-arrival ones.

3.2.1 Pruning (J, R^{δ}_{J}) Pairs with Topology Affinity

The released resources R^Δ can be tens to hundreds of GPUs when a large job finishes; an ongoing job J does not necessarily take all the idle resources—it may benefit most when allocated R_J^δ, *a subset of* R^Δ; and partitioning R^Δ over multiple jobs may achieve better overall efficiency. However, the number of subsets of R^Δ grows exponentially with $|R^\Delta|$, and respectively working out the parallelization plan for each (J, R_J^δ) is cost-prohibitive. Therefore, we need to first safely reduce the number of (J, R_J^δ) pairs to explore.

With incremental reconfiguration, *Suika* uses the current allocation R_J^0 as reference to select only an array of promising subsets of R^Δ, by *enforcing a rigid resource provisioning order based on the topology affinity*. The intuition is that, higher affinity yields higher communication bandwidth, and more likely higher throughput and scaling efficiency. Therefore, as shown in Fig. 8, given a set of available GPUs R^Δ, job J would always prefer to take closer GPUs to its current ones (better within the same node or rack).

Specifically, given a running job J with its current resource allocation R_J^0, *Suika* sorts the available resources R^Δ in descending order with respect to $\text{Affinity}(r^\Delta, R_J^0) = \max_{r^0 \in R_J^0} \text{EffectiveBandwidth}(r^\Delta, r^0)$, resulting in an ordered list $\hat{R}_J^\Delta = \left[\hat{r}_1^\Delta, \hat{r}_2^\Delta, ..., \hat{r}_{|R^\Delta|}^\Delta\right]$ [4]. The candidate resource subsets R_J^δ are restricted to *prefixes of this list* $R_i^\delta = \left\{\hat{r}_1^\Delta, \hat{r}_2^\Delta, ..., \hat{r}_i^\Delta\right\}, i \in [|R^\Delta|]$. This heuristic reduces the *exponential* space of resource allocations for a job to $|R^\Delta|$ possibilities, and naturally promotes locality and defragmentation. For clarity, let $\bar{P}_i$ represent the best parallelization plan for (J, R_i^δ) with allocation $R_J^0 \cup R_i^\delta$, and view P_J^0 as $\bar{P}_0$. Next, we explore how to efficiently calculate $\bar{P}_i$ for each (J, R_i^δ) pair.

3.2.2 Incremental Parallelization for Each (J, R_J^δ) Pair

Though having reduced the number of (J, R_J^δ) pairs to $|R^\Delta|$, for each (J, R_J^δ) pair, naively applying solver like *Metis* to get the optimal parallelization plan remains cost-prohibitive. Motivated by our analysis in §2.3, *Suika* seeks to tackle such complexity by exploiting the incremental nature of rescheduling. That is, when a running job is to receive additional resources, *Suika* preserves its current *parallelization skeleton* (including number of PP stages, respective TP/DP strategy for each stage, mapping of ranks to physical GPUs), and incrementally expands it with new resources along the PP, DP or TP dimensions—with a dynamic programming approach formalized in Alg. 1.

Efficient 3D-parallelization planning in an *expand-and-balance* manner. *Suika* adopts an *expand-and-balance* strategy for atomic incremental parallelization, as described in the `3D-Expand-and-Balance()` function. To illustrate, consider deriving P_1 with the base plan P_0 and additional resources R. The `PP/DP/TP-Expand()` functions generate candidate plans by (1) adding R as a new pipeline stages with multiple TP/DP-degree attempts, (2) merging R to increase a stage's DP degree, and (3) merging R to vary a stage's TP degree. These local adjustments of parallelization skeleton necessarily introduce workload-imbalance. Thus, a load-balance procedure (line 20) is applied to all candidates. It solves a weighted min-max bottleneck problem to determine the per-stage layer number and per-DP-group local batch size to minimize stragglers (details in Appendix B), guided by the performance model in §3.1. After the balancing process, the candidate plan that yields the highest throughput is selected and returned.

Algorithm 1 Incremental Plan Solver

1: **function** GetBestRsAndThps(J, R^Δ)
2:     // Get preferred allocations and throughputs w./ best plans
3:     $\hat{r}^\Delta_{[1..|R^\Delta|]} \leftarrow$ sort R^Δ by topology affinity to R_J^0 [5]
4:     $\bar{P}_0 \leftarrow P_J^0$
5:     **for** $i \leftarrow 1$ to $|R^\Delta|$ **do**
6:         $\bar{P}_i \leftarrow \text{incrementalParallelize}(\bar{P}_{[0..i-1]}, \hat{r}^\Delta_{[1..i]})$
7:         $R_i^\delta \leftarrow \hat{r}^\Delta_{[1..i]}, \bar{\text{T}}_i \leftarrow \text{Throughput}(\bar{P}_i)$
8:     **return** $R^\delta_{[1..|R^\Delta|]}, \bar{\text{T}}_{[1..|R^\Delta|]}$
9: **function** incrementalParallelize($\bar{P}_{[0..i-1]}, \hat{r}^\Delta_{[1..i]}$)
10:     $\mathcal{P}_{\text{candidate}} \leftarrow \emptyset$
11:     **for** $j \leftarrow 1$ to i **do**
12:         $P \leftarrow \text{3D-Expand-and-Balance}(\bar{P}_{i-j}, \hat{r}^\Delta_{[i-j+1..i]})$
13:         $\mathcal{P}_{\text{candidate}} \leftarrow \mathcal{P}_{\text{candidate}} \cup \{P\}$
14:     **return** $\arg\max_{P \in \mathcal{P}_{\text{candidate}}} \text{Throughput}(P)$
15: **function** 3D-Expand-and-Balance(P_0, R)
16:     $\mathcal{P}_{\text{expand}} \leftarrow \emptyset$
17:     $\mathcal{P}_{\text{expand}} \leftarrow \mathcal{P}_{\text{expand}} \cup \text{PP-Expand}(P_0, R)$
18:     $\mathcal{P}_{\text{expand}} \leftarrow \mathcal{P}_{\text{expand}} \cup \text{DP-Expand}(P_0, R)$
19:     $\mathcal{P}_{\text{expand}} \leftarrow \mathcal{P}_{\text{expand}} \cup \text{TP-Expand}(P_0, R)$
20:     $\mathcal{P}_{\text{expand-balance}} \leftarrow \{\text{Load-Balance}(P) | P \in \mathcal{P}_{\text{expand}}\}$
21:     **return** $\arg\max_{P \in \mathcal{P}_{\text{expand-balance}}} \text{Throughput}(P)$

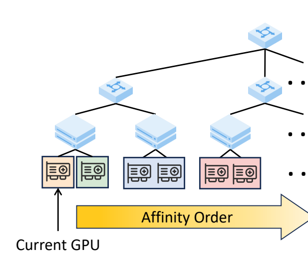


Figure 8. Resource provisioning in affinity order.

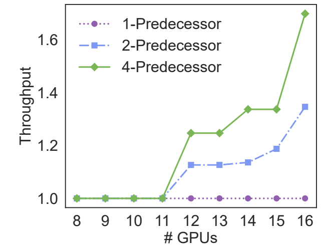


Figure 9. Considering more predecessors improves plan.

[4] The context throughout §3.2.1 to §3.2.2 is always job-specific, so we omit subscript J when it conflicts with index subscript.
[5] The sorting has a secondary key—(rack id, node id, GPU id)—to group new resources with better affinity to each other together.

Ensuring optimality with multi-predecessor incremental parallelization. The expand-and-balance procedure serves to atomically derive P_1 with fixed P_0 and R. Intuitively, one can progressively get $\bar{P}_i$ out of $\bar{P}_{i-1}$ by adding one GPU $\hat{r}_i^{\Delta}$. However, this method easily falls into local optima. In contrast, to calculate $\bar{P}_i$, *Suika* considers all the possible predecessors $\bar{P}_0, ..., \bar{P}_{i-1}$ to enable adjustments at diverse granularities, as depicted in the `incrementalParallelize()` function.

To demonstrate its necessity, we conduct an experiment (also with our testbed in §5.1) that expands a LlaMA2-7B job from 8 GPUs (one node) to 16 GPUs (two nodes). We respectively set the predecessor window size to 1, 2 and 4; for example, 2 means that when calculating a 16-GPU plan, we select the best from "adding 2 GPUs to a 14-GPU plan" and "adding 1 GPU to a 15-GPU plan". Fig. 9 shows the best throughput attained during that expansion process in each case; it reveals that the 4-predecessor method finds better plan than 2-predecessor, meanwhile 1-predecessor fails to find any plan to improve throughput. Such a result can demonstrate the need to enable multi-predecessor expansion.

With the previous techniques, we obtain the full knowledge on how a job's throughput varies with its resource allocation amount—provided that the GPUs are provisioned following the topology-affinity order, i.e., in an inclusion chain $R_1^{\delta} \subseteq R_2^{\delta} \subseteq ... \subseteq R_{|R^{\Delta}|}^{\delta} = R^{\Delta}$. Such knowledge is then provided to the *SuikaSched* to make allocation decisions.

Time complexity and pruning. Let $M = |R^{\Delta}|$, $N = |R_J^0|$, and assume other constants related to model architectures are bounded constants (collectively contribute a multiplier around 10 in practice), then the overall time complexity is $O(M^2(N + M))$. This is directly applicable for $N, M \sim 10^2$ under seconds.

A few additional pruning techniques can be applied in extreme cases: (1) If $M >> N$, we may early-stop at $M \approx N$, as expanding a job significantly beyond its current allocation typically yields diminishing returns in utilization. (2) When M is large, it is not necessary to explore all predecessors. We may limit the window to closest W ones. (3) When N is large, it is not necessary to adjust at single-GPU granularity. We may group every w intra-node GPUs as the minimum unit.

Equipped with these pruning, the complexity becomes $O(NMW/w)$, and scalable to $N, M \sim 10^3$ with $W = w = 8$. We empirically set $W = 8$ and $w = \min\{\lceil N/16 \rceil, 8\}$, which delivers decision latency under seconds while nearly always matching the schedule quality of the unpruned baseline.

For a detailed complexity analysis, please see Appendix B.

3.2.3 Inter-Job Scheduling

Apart from the previous intra-job innovations, a practical scheduler must manage multiple concurrent jobs and prepare for future jobs. This introduces two inter-job scheduling challenges: (1) how to partition available resources among multiple running jobs to maximize overall throughput gain, and (2) how to balance accelerating running jobs against provisioning resources for future jobs.

Algorithm 2 Inter-job Scheduling

1: **function** SCHEDULE($\mathcal{J}_{\text{wait}}$, $\mathcal{J}_{\text{run}}$, R^{Δ})
2: scheduleWaitingJobsWithPreempt($\mathcal{J}_{\text{wait}}$, $\mathcal{J}_{\text{run}}$, R^{Δ})
3: reclaimScaleOutBelowTHR($\mathcal{J}_{\text{run}}$, R^{Δ})
4: **while** $R^{\Delta} \neq \emptyset$ **do**
5: $(J,\ R^{\delta},\ B) \leftarrow$ getBestScaleOut($\mathcal{J}_{\text{run}}$, R^{Δ})
6: **if** $B < \tau(\mathcal{J}_{\text{run}})$ **then break**
7: applyScaleOut(J, R^{δ})
8: $R^{\Delta} \leftarrow R^{\Delta} \setminus R^{\delta}$
9: **function** getBestScaleOut($\mathcal{J}_{\text{run}}$, R^{Δ})
10: **for** $J \in \mathcal{J}_{\text{run}}$ **do**
11: $\delta_J^{[1..|R^{\Delta}|]}, \mathrm{T}_J^{[1..|R^{\Delta}|]} \leftarrow$ getBestRsAndThps(J, R^{Δ})
12: $\bar{J},\ \bar{i} = \arg\max_{J \in \mathcal{J}_{\text{run}},\ i \in |R^{\Delta}|} \mathrm{B}(R_J^0,\ \delta_J^i,\ \mathrm{T}_J^0,\ \mathrm{T}_J^i)$
13: $\bar{R}^{\delta} \leftarrow\ \delta_{\bar{J}}^{\bar{i}},\ \bar{B} \leftarrow \bar{\mathrm{B}}(R_{\bar{J}}^0,\ \bar{\delta}_{\bar{J}}^{\bar{i}},\ \mathrm{T}_{\bar{J}}^0,\ \mathrm{T}_{\bar{J}}^{\bar{i}})$
14: **return** $(\bar{J},\ \bar{R}^{\delta}, \bar{B})$

Multi-round scale-out to partition R^{Δ} with best marginal benefits. To measure the scaling efficiency across jobs, *Suika* adopts a marginal benefit metric B, formally defined as:

$$\mathrm{B}(R^0, R^{\delta}, \mathrm{T}_{R^0}, \mathrm{T}_{R^0 \cup R^{\delta}}) = \frac{\|R^0\|}{\|R^{\delta}\|} \left(\frac{\mathrm{T}_{R^0 \cup R^{\delta}} - \mathrm{T}_{R^0}}{\mathrm{T}_{R^0}} \right).$$

The B essentially characterizes $\frac{1}{\|R_J^{\delta}\|}(\|R_J^0\|\bar{\mathrm{T}}_{\text{norm}}(J, R_J^0, R_J^{\delta}) - 1)$, where the weighting terms $\|R_J^0\|$ and $\frac{1}{\|R^{\delta}\|}$ are included for the same reason discussed in Footnote 1.

Equipped with B, *Suika* allocates spare GPUs based on marginal benefit maximization, a well-accepted strategy in scheduling contexts [15, 50, 79] due to the scheduling problem's NP-hardness.

Technically, given R^{Δ}, ① *SuikaSched* first collects each job's preferred resource allocations as well as the optimal-throughput for each allocation with Alg. 1; the best job-allocation pair $(\bar{J}, \bar{R}^{\delta})$ with global optimal marginal benefit B is applied (line 9 ∼ 14 in Alg. 2); ② *SuikaSched* deducts selected $\bar{R}^{\delta}$ from R^{Δ} and repeats the process until no efficient scale-out remains (line 4 ∼ 8). This multi-round procedure ensures R^{Δ} is sufficiently utilized and properly distributed to multiple jobs with best marginal benefits.

Balance between scaling-out running jobs and serving future jobs. *Suika* adopts a two-tier resource demand classification inspired by [33]: (1) *basic demand*, which corresponds to the minimum resources explicitly requested by the user to launch the job (as they would in a non-elastic setting), and (2) *expansive demand*, which represents any additional resources allocated elastically to accelerate training. To prevent elasticity from hindering new submissions, *Suika* enforces a strict priority: *future jobs' basic demand always takes precedence over existing jobs' expansive demand* (line

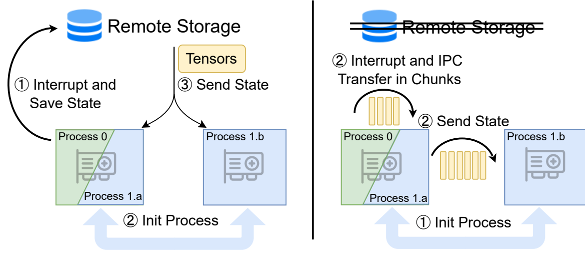


Figure 10. Comparison between checkpoint-resume redeployment (left) and our incremental redeployment (right).

2). When a new job arrives, ① *Suika* tries to provision its basic demand; ② if infeasible, it repeatedly reclaims scale-out with the lowest B (*Suika* records history of all scale-outs) and retries; ③ if still infeasible, it occasionally applies defragmentation as the last resort. This ensures that new jobs' basic demand will be satisfied under *Suika*, as long as it is satisfiable without elasticity. Note that the exact policy to schedule queuing jobs (e.g., FIFO or SJF) is decoupled from *Suika*, and exposed as a flexible API.

Moreover, for future jobs' expansive demand — to transfer expanded resources from current jobs to future jobs with higher efficiency — *Suika* adopts an *at-least-as-good-as* principle: all scale-outs must have B over a threshold of $\tau = U^{\lambda_\tau}$, where U depicts the cluster-wide resource intensity (i.e., dividing the sum of all ongoing jobs' basic demands by the total cluster resources), and λ_τ is knob controlling the expansion aggressiveness. With this threshold, all scale-outs are sufficiently efficient and frequent allocation-and-reclamation oscillations are avoided. Further, when cluster load rises, scale-outs below τ will be reclaimed and release their expanded resources into rescheduling pool (line 3).

Finally, to minimize disruption, Suika employs batch reconfiguration: it simulates multiple incremental decisions in a virtual environment, merges the intermediate steps, and applies only the final consolidated plan to the cluster, thereby eliminating the overhead of multi-iteration reconfigurations.

3.3 Redeployment Optimization

Upon a rescheduling decision, the selected jobs must promptly switch to the new parallelization plan, which requires interruptive job redeployment. The redeployment overhead consists of two parts: (1) *initialization* of environment on new workers, including CUDA context, NCCL communicator setup, etc., and (2) *migration* of training state to new workers, including model parameters and optimizer states.

For *initialization*, a classic technique is to hide (overlap) initialization with the training; training is interrupted only after initialization completes [31, 72, 74]. For *migration*, noticing the inefficiency of naively checkpointing the full state to remote storage, a recent work, *Tenplex* [67], provides a device-to-device migration mechanism to minimize data movement. *Suika* integrates these techniques, applying both initialization overlapping and device-to-device migration.

Taking a step further, *Suika* exploits an unique property of incremental rescheduling: parallelization plans across reconfiguration are highly similar, meaning most states already reside on their target GPUs (referred to as *local state*). However, as designed for generic scenarios, *Tenplex* adopts a CPU-centric storage, which incurs GPU-CPU transfer and CPU-side serialization overhead even for local state. Instead, *Suika* specializes intra-GPU transfer with *CUDA IPC (inter-process communication)* to copy local state directly on target GPU, bypassing offloading and serialization. This reduces local state migration time by an order of magnitude, to only seconds with 20 GB data.

Moreover, performing migration entirely on GPU leads to extra memory overhead, risking Out-Of-Memory. To address that, *Suika* manages the life-cycle of state tensors with extreme caution, and adopts *chunked transfer*: the state is sent via CUDA IPC in chunks, and the previous worker releases a chunk as soon as new worker receives and clones it. As summarized in Fig. 10, *Suika* combines initialization overlap, device-to-device migration, and CUDA-IPC to speedup the redeployment, and chunked transfer to avoid OOM.

4 Implementation

We have implemented *Suika* from scratch with a total of 9k LoC: 4k Python code for *SuikaSched*, 1k C++ code for *SuikaAgent*, and 4k Python code for *SuikaEngine*.

Specifically, in *SuikaSched* we implement all the core scheduling functionalities such as resource tracking, scheduling policies and job status monitoring. The *SuikaAgent* encapsulates the core Incremental Plan Solver algorithm (Alg. 1) and the performance model. For better scalability and low latency, the algorithm is implemented in C++ and exposed to Python with `pybind11` [51], pinning each *SuikaAgent* to a CPU core. The *SuikaEngine* is central for enabling adaptive parallelization plan execution. It supports asymmetric 3D-parallelism by automatically slicing user models along DP and TP dimensions and orchestrating the PP data flow. The engine is built on top of basic *PyTorch* [52] functionalities, and is compatible with *HuggingFace* [22] LLM models. Meanwhile, to enable online performance modeling, *SuikaEngine* conducts user-transparent full-spectrum profiling by inserting *CUDA events* [48] for high-resolution timing without introducing additional synchronization overhead; it also implements the redeployment optimization techniques described in §3.3.

While we implemented *Suika* from scratch to maximize flexibility during the exploration, its design is inherently modular, with the scheduler logic decoupled from the training framework. This allows our runtime engine to be replaced with industry-standard frameworks [56, 57] with appropriate and manageable modifications.

Type	Model	PP-DP-TP	Type	Model	PP-DP-TP
S	GPT2-350M	1-1-1	XL	GPT2-39B	2-4-4
	GPT2-1.3B/2.6B	1-4-1		LLaMA-30B	2-4-4
	Qwen2-0.5B	1-1-1		GPT2-76B	2-4-8
	Qwen2-1.5B	1-4-1		LLaMA-65B	2-4-8
M	GPT2-6.7B	1-4-2		Qwen2-72B	2-4-8
	LLaMA2-7B	1-4-2		LLaMA2-70B	3-8-8
	Qwen2-7B	1-4-2		GPT2-175B	4-8-8
L	GPT2-15B	1-4-4		(Simulation Only)	
	LLaMA2-13B	1-4-4			

Table 1. LLMs (with default 3D plans) for our experiments.

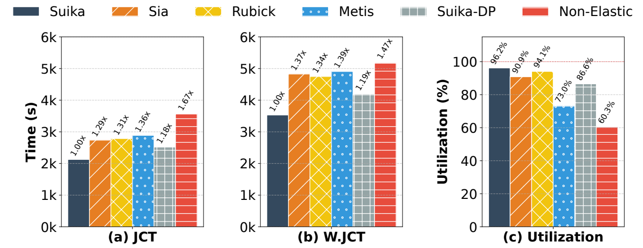


Figure 11. End-to-end performance with Cluster-A.

5 Evaluation

5.1 Experiment Setup

Hardware platform. We conduct testbed evaluations atop a 64-GPU Cluster (Cluster-A) consisting of 8 nodes, where each node has 128 CPU cores, 1 TB CPU RAM and 8 NVIDIA H100 GPUs (interconnected with 900 GB/s NVLink). For remote connection, each node has 8 InfiniBand NDR NICs (400 Gbps each), plus 1 InfiniBand NDR NIC dedicated for storage I/O; all nodes are connected under a single rack with multi-rail leaf-spine topology. Additionally, to evaluate *Suika* performance at a larger scale, we simulate another cluster (Cluster-B) comprising 128 nodes and 1024 GPUs—replicating the per-node configuration of Cluster-A but distributed across multiple racks.

Workloads. As listed in Table 1, we choose three representative model families[6]: GPT [3, 54], Qwen [61] and LLaMA [62, 63], each with multiple model volumes—from 350M to 15B in testbed experiments (no larger due to our relatively limited cluster scale), and further to 175B in simulations (with `XL` type included). For each model specification, the initial plan is tuned to optimal under a fixed number of GPUs and symmetric 3D-parallelism. Following the *Rubick* work [79], we select the busiest 8-hour period in the Microsoft *Philly* trace [26] (busier periods are more challenging for *Suika* to attain good performance), and down-sample jobs proportional to cluster size. For each job entry in the trace file, we randomly map it to a LLM training job in Table 1, with a selection ratio of $S : M : L = 3 : 1 : 1$; for that job we preserve its submission time and scale training iterations by a global factor so that total GPU-time consumption is identical with that of the traced one.

Baselines. In our evaluation, we set (1) the non-elastic scheduler First-In-First-Out as a basic baseline. Besides, we set (2) *Sia* [25], (3) *Rubick* [79] and (4) *Metis* [64] as three additional baselines[7], all introduced in §2. Moreover, for ablation study we also evaluate (5) *Suika*-DP, a customized version of *Suika* with 3D parallelization degraded to 1D.

Hyper-parameters. For the hyper-parameters in those baseline methods, we in general follow the suggested setup in their original papers or open artifacts. Specifically, for *Metis*, the `min_group_scale_variance` is set to 0.5, and the `max_permute_len` hyperparameter is set to 4; besides, we also set a searching time-budget of 60s, beyond which the searching results are not considered. For *Sia*, the fairness knob p_{fairness} is set to -0.5, the waiting job penalty λ_n is set to 1.1, and the scheduling interval is set to 60s. For *Rubick*, the reconfiguration penalty threshold is set as $k_{\text{reconfig}} = 0.97$, and the queueing penalty threshold is set as $k_{\text{queue}} = 200s$. For *Suika*, we set the expansion aggressiveness knob λ_τ to 1.0, and the base queueing policy to First-In-First-Out, same as default non-elastic scheduler.

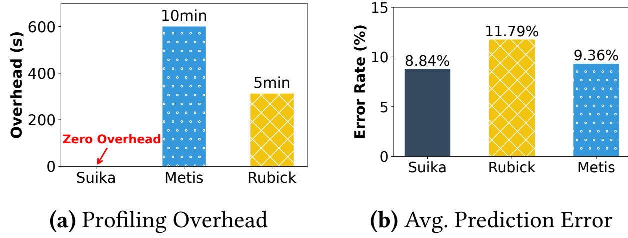
Metrics. We compare *Suika* against baselines with three metrics: (1) average Job Completion Time (JCT); (2) average Weighted Job Completion Time (W.JCT), where a job's basic GPU demand is used as the weight for calculating average JCT (consistent with the objective form shown in Eq. 1); (3) Cluster Utilization, which is the average ratio of allocated GPUs over the entire experiment period (i.e., total GPU-time divided by makespan).

5.2 End-to-end Performance

Fig. 11 shows the end-to-end performance of *Suika* against the other baselines. When compared with the default non-elastic scheduler, *Suika* can reduce average JCT and W.JCT by 1.67× and 1.47×, respectively; regarding cluster utilization, it makes an improvement of 59.5%. Such results confirm that it is highly rewarding to enable runtime resource rescheduling

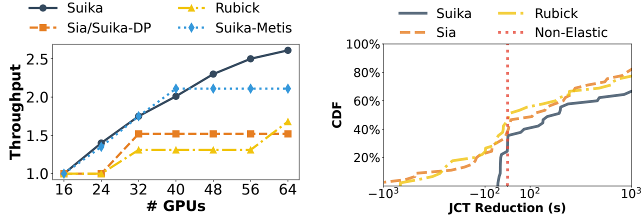
[6]While newer models [11, 12, 14, 76] keep emerging rapidly, the models we select suffice for verifying *Suika* effectiveness: such selected models and the newer-generation ones are all composed of a series of identical Transformer-based layers, thus sharing the same 3D-parallel training characteristics.

[7]For *Metis*/*Sia*/*Rubick*, we offline profile sufficient data to bootstrap their performance model. We set jobs as strong-scaling for *Sia*, and disable ZeRO for *Rubick*, as those two dimensions are orthogonal to *Suika* (discussed in Sec. 7). *Metis* is relatively fast and of high-quality among 3D-parallelism solvers [39, 64, 65, 81]; we choose it as the representative and integrate it with the high-level scheduling logic of *Suika* (replacing `incrementalParallelize()` function with *Metis*, solving for each (J, R_J^δ) in parallel under a time budget.).



(a) Profiling Overhead **(b)** Avg. Prediction Error

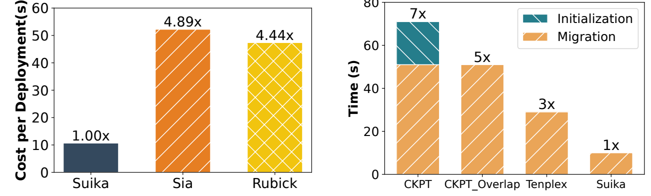
Figure 12. Comparisons on performance modeling.



(a) Plan quality of LlaMA2-13B as resource expands.

(b) Impact on individual jobs relative to the non-elastic scheduler.

Figure 13. Comparisons on decision quality.



(a) Per-redeployment time cost under different schedulers.

(b) Cost break-down under different redeployment methods.

Figure 14. Comparisons on redeployment cost.

in shared clusters (with global batch size preserved, this does not affect the model accuracy, as confirmed in [5, 79]).

We next compare *Suika* with state-of-the-art elastic schedulers. When compared to *Sia* and *Rubick*, *Suika* reduces the average JCT by 1.29× and 1.31×, respectively; for average W.JCT, *Suika* achieves even greater improvements of 1.37× and 1.34×. Notably, although *Sia* and *Rubick* achieve high utilization similar to *Suika*, this does not translate to high efficiency with their suboptimal parallelization plans. Comparing to *Metis*, *Suika* delivers 1.36×, 1.39× improvement in JCT and W.JCT (*Metis* has much lower utilization because it may fail to yield effective parallelization plans to expand large jobs within the time budget). In general, *Suika* outperforms these schedulers because of its capability[8] to make fast-yet-high-quality reschedule decisions and light-weight redeployments.

For ablation study, Fig. 11 further shows that, relative to the *Suika-DP* variant, *Suika* gains 1.18×, 1.19× improvement in JCT and W.JCT, confirming the benefits of leveraging the full 3D solution space. Notably, *Suika-DP* alone attains 1.09× JCT reduction over *Sia*, attributed to faster redeployment with incremental configuration.

5.3 Microscopic Performance Deep Dive

In this part, we respectively inspect the superiority of *Suika* in profiling, decision-making and redeployment.

Performance modeling superiority. We compare *Suika* against two existing 3D modeling methods, *Rubick* and *Metis*, with respect to profiling overhead and prediction accuracy. As in Fig. 12a, *Suika* incurs *no extra overhead* with non-disruptive full-spectrum profiling; in contrast, *Rubick* spends 6 minutes bootstrapping each job's performance model as over multiple configurations and *Metis* incurs an average of 10 minutes overhead per job.

For prediction accuracy, we evaluate each method on the set of plans it selected during end-to-end experiment, ensuring that the plans lie within the method's intended domain. As in Fig. 12b, *Suika* attains even better prediction accuracy (8.84% average error) over *Rubick* (11.79%) and *Metis* (9.36%). We observe that *Suika*'s prediction error stems primarily from variations in GPU utilization between profiled and predicted plans, whereas factors such as model size or specific parallelization strategies exhibit minor impact. Such errors are usually negligible for filtering effective plans. To justify this, we replay the trace simulation with injected Gaussian error $\sim \mathcal{N}(\mu = 0, \sigma = \epsilon)$; the average JCT increases by only 2% and 6% for $\epsilon = 10\%$ and 20%, respectively. This resilience stems from the substantial throughput gaps between optimal and suboptimal plans, which mask moderate prediction inaccuracies.

Decision quality superiority. We conduct a microscopic measurement to verify the parallelization plan quality of *Suika*. Specifically, for a LlaMA2-13B job, we incrementally increase its GPU allocation from 16 to 64, and compare the parallelization plans derived under different methods: *Suika*, *Metis*, *Rubick*, and *DP-only* (shared by *Sia* and *Suika-DP*). As shown in Fig. 13a, *Suika* consistently identifies plans comparable to *Metis*, both substantially outperforming the DP-only (*Sia*) and the symmetric-3D (*Rubick*) baselines. Moreover, *Metis* fails to produce solutions beyond 40 GPUs due to solver timeouts, while *Suika* continues to scale effectively. When provided with unlimited search time, *Metis* offers only a marginal throughput gain (2%-6% for 40-64 GPUs) over *Suika*, but at the cost of significant search latencies ($\geq$ 5 min vs. $\leq$ 1 s).

Moreover, recall that in *Suika* we adopt a principle of *basic-allocation-first* to prevent the expansion of ongoing jobs from delaying future-arriving ones (§3.2.3). To confirm the benefit of such principle, we set the non-elastic scheduler as the performance baseline, and calculate the JCT reduction

[8] Note that for *Rubick* and *Metis*, we *do not* count the disruptive offline profiling overheads in JCT (further described in §5.3)—counting them would only amplify *Suika*'s advantage.

Figure 15. Experimental results in sensitivity analysis.

of each job respectively under *Suika*, *Sia* and *Rubick*; the CDFs are depicted in Fig. 13b. It shows that *Suika* delays jobs by at most 60s (while attaining a large overall speedup); in contrast, *Sia* and *Rubick*, by being throughput-centric and overly-penalizing redeployment, often cause substantial execution delay for individual jobs.

Redeployment efficiency superiority. *Suika* can remarkably speedup LLM redeployment. In Fig. 14a we depict the average per-redeployment cost during the end-to-end experiments. It shows that *Suika* makes an speedup of 4.89× and 4.44× when compared with *Sia* and *Rubick*.

We further make an ablation study on the effectiveness of each redeployment optimization technique. We measure the redeployment time for LlaMA2-13B from 16 to 24 GPUs with gradually stacked optimizations: (1) CKPT—the default checkpoint-resume method, (2) CKPT_Overlap — enhanced with initialization overlap, (3) Tenplex—further enhanced with device-to-device transfer, (4) *Suika*—enhance Tenplex with CUDA-IPC. Fig. 14b shows that each technique yields a substantial overhead reduction. We also note that our chunked transferring technique (with a chunk size of 2GB) avoids OOM errors when redeploying large models like LlaMA2-13B and GPT2-15B—which would have happened otherwise.

5.4 Sensitivity Analysis

To assess *Suika* robustness, we conduct a series of experiments under diverse setups, and Fig. 15 shows the results.

(a) Inter-node bandwidth. Network conditions affect the scaling efficiency after resource expansion. To evaluate *Suika* performance under diverse network conditions, we respectively create three scenarios: (1) Full-IB (8×400 Gbps)—the default testbed, (2) Half-IB (4×400 Gbps)—by disabling half IB NICs in a way that every 2 GPUs are affine to one NIC, and (3) No-IB (10 Gbps)—by disabling all IB NICs and falling back to Ethernet.[9] Fig. 15a confirms that *Suika* consistently achieves the lowest average JCT across all the scenarios (the benefits under *Suika* are lower in No-IB case because the resource expansion space is more constraint).

(b) Job intensity. To evaluate *Suika* benefit under diverse load intensities, as shown in Fig. 15b, we respectively scale down the job submission intervals by different ratios (divided respectively by 0.75, 1, 1.25, 1.5 and 2). Fig. 15b suggests that *Suika* makes the best JCT in each case, with larger advantages in lighter-loaded cases because they present more chances for 3D-parallel scaling.

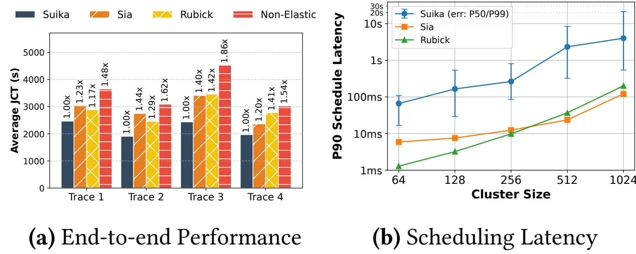
(c) Job composition. Recall that in our default workload, the ratio of small (S), middle (M) and Large (L) jobs is 3:1:1; we then check whether different job composition affects the performance of *Suika*. To evaluate, we group M and L jobs together and change the S:(M+L) ratio from 3:2 respectively to 4:1 and 2:3. As shown in Fig. 15c, *Suika* can work consistently well in each case.

(d) Expansion-aggressiveness knob λ_τ. We vary the λ_τ between 0.25 and 2.0. As shown in Fig. 15d, *Suika*'s performance remains similar as long as λ_τ is not over-conservative, where less than 5% difference in average JCT is observed for $\lambda_\tau \geq 0.75$. This robustness arises because *Suika* consistently identifies sufficiently high-efficiency plans in the present high-bandwidth setup.

(e) Base queuing policy. To prove that *Suika* generalizes to different base queueing policy, we replace FIFO with SJF for *Suika*. Results in Fig. 15e show that *Suika* achieves similar speedup (1.73×) when applied atop SJF.

(f) Trace. We evaluate *Suika* performance with three additional traces—one from Philly [26], and two from Helios [18]

[9] For No-IB case, initial plan involving inter-node data parallelism is changed to pipeline parallelism.



(a) End-to-end Performance **(b)** Scheduling Latency

Figure 16. Results in large-scale simulations.

(with higher contention). Trace Philly1 is previously adopted in the end-to-end experiment, and Philly2 is another 8-hour trace sampled randomly. Fig. 15f shows that *Suika* consistently outperforms others across all the cases.

5.5 Results in Large Scale Simulation

We resort to simulations to evaluate *Suika* at a larger scale. We build a discrete-time, event-driven simulator directly atop *Suika*'s physical execution framework: it shares the identical performance model and scheduling logic with the physical testbed, while emulating the training execution. Our fidelity test with previous testbed workloads shows that, in average JCT there is a maximum gap of 3% between the simulation and testbed experiment. The error is comparable to simulators in previous works [25, 53, 79] and we deem it acceptable.

In our simulation, we scale the cluster size to 1024 GPUs, trace to 24 hours, and jobs to up to 175B (256 GPUs). To mimic a multi-level network topology, every 16 nodes are placed under the same rack, and cross-rack bandwidth is discounted by 50% to simulate network congestion [36].

Regarding end-to-end performance, evaluation on four 24-hour traces (Fig. 16a) shows *Suika* achieves similar speedup as on physical testbed, on average attaining 1.63×, 1.32× and 1.32× JCT reduction over *Non-Elastic*, *Sia* and *Rubick*. This proves *Suika*'s decision quality scales well to large clusters.

To evaluate *Suika*'s decision speed as cluster size increases, we plot the P90 scheduling algorithm latency for *Suika*, *Sia*, and *Rubick* in Fig. 16b. For *Suika*, we also show P50 and P99 values as error bars. Although *Suika* is slower than *Sia* and *Rubick* (which is reasonable since it explores a much larger solution space to ensure plan optimality), it still achieves practical latency: P90 at 4.01 seconds on a 1024-GPU cluster, P50 at 0.54 seconds, and P99 at 21.58 seconds.

6 Additional Related Work

Parallelization schemes for LLMs. With the growing adoption of mixture-of-experts (MoE) [24, 30] architectures and long-context training [6], conventional 3D-parallelism (elaborated in §2) has been extended to *5D-Parallelism* with *expert parallelism* (EP) [30, 58] and *sequence parallelism* [28, 35] (SP). Parallelism techniques to save GPU memory such as ZeRO [55] are also widely used. We cover how *Suika* can be adapted to support them in §7.

GPU cluster scheduling. Various GPU cluster schedulers have been previously proposed with diverse objectives, like for reducing job completion time [16, 18–20], improving cluster utilization [23, 43, 73] and guaranteeing fairness [4, 41, 80]. Nonetheless, these schedulers mainly focus on determining the job queuing order or initial resource allocation, ignoring how to make resource rescheduling after a job has started. In that sense, they can be combined with *Suika* for better performance (as already suggested in Fig. 15e.)

Redeployment optimization. In fault tolerance domain, overlapping checkpointing with training has been extensively studied [42, 46, 68, 69, 71]; Deepspeed's UCP [38] introduces a universal checkpoint representation for diverse parallelism strategies. In runtime initialization, existing works have also proposed to overlap the runtime initialization [72, 74] as well as the code-start delay [29] with training. However, these works fail to exploit the property of incremental redeployment (an opportunity provided only by *Suika*), thereby suffering suboptimal redeployment efficiency.

7 Discussions

Extending to 5D-Parallelism and ZeRO. *Suika* generalizes naturally to 5D-Parallelism and ZeRO: (1) non-disruptive performance modeling can be extended to new parallelism (e.g., by collecting the routing statistics for EP which are generally stable [32]); (2) the *topology-affinity-based resource ordering* and *expand-and-balance* principles in *Suika* are by nature extensible for scenarios with additional parallelization dimensions; (3) the redeployment framework operates on abstracted state representations, requiring only minor extensions to handle parameter sharding under ZeRO. We plan to enforce such extensions in future work.

Fairness. While primarily optimizing cluster efficiency, Suika can support fairness-aware policies by discounting the marginal benefits of jobs that have consumed disproportionate elastic resources. With appropriate discounting functions, Suika can align scheduling decisions with diverse fairness objectives.

Hyperparameter tuning. To balance efficiency and fairness, operators can leverage Suika's high-fidelity simulator to costlessly sweep and tune hyperparameters like expansion aggressiveness (λ) for specific workloads and policy goals.

Convergence validity and global batch size tuning. *Suika* does not modify global batch size across reconfigurations, thereby preserving model convergence [5, 79]. Prior works for traditional deep learning [25, 53] incorporates global batch size tuning as a scheduling dimension to maximize goodput (throughput × statistical efficiency, adopted in *Pollux* [53] and *Sia* [25]). In principle, with a statistical efficiency estimation function, *Suika* can replace *throughput* with *goodput* to enable batch elasticity for faster convergence.

Failure recovery. Failure recovery during LLM training is orthogonal to Suika's design, and standard checkpointing system[68, 69, 71] can be integrated seamlessly. Upon job failure, Suika can temporarily reserve the allocated resources to wait for external recovery processes rather than immediately reclaiming them.

Adapting to super-large clusters. While we scale out to 1,024 GPUs in simulation, pioneering industrial practitioners have reported maintaining 10,000 or even more GPUs in their cluster [27]. Naively applying *Suika* in such scenarios still yields a relatively large solution space. We recommend dividing the original cluster into multiple rescheduling sub-zones, or increase the minimum rescheduling unit from individual GPUs to nodes or even a couple of nodes based on $|R_J^0|$.

8 Conclusion

In this paper, we propose *Suika*, a holistic system that enables both efficient and high-quality rescheduling for 3D-parallelized LLM training jobs. *Suika* exploits the opportunities of incremental job reconfiguration to optimize performance modeling (with non-disruptive full-spectrum profiling), decision-making (with affinity-based pruning and expand-and-balance planning), and redeployment (with CUDA IPC based transferring). Experimental results show that *Suika* can improve the average JCT by over 1.29× when compared with state-of-the-art elastic schedulers.

Acknowledgments

We sincerely thank our shepherd, Sahil Parmar, and all the anonymous reviewers for their valuable feedback. We also thank Huawei Cloud and TeleAI for their supports. This work is partially funded by National Natural Science Foundation of China (NSFC) under grant No. 62572304 and 62232011.

References

[1] Hongjun An, Wenhan Hu, Sida Huang, Siqi Huang, Ruanjun Li, Yuanzhi Liang, Jiawei Shao, Yiliang Song, Zihan Wang, Cheng Yuan, et al. Ai flow: Perspectives, scenarios, and approaches. *Vicinagearth*, 3(1):1, 2026.

[2] Zhengda Bian, Shenggui Li, Wei Wang, and Yang You. Online evolutionary batch size orchestration for scheduling deep learning workloads in gpu clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

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

[4] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[5] Chen Chen, Qizhen Weng, Wei Wang, Baochun Li, and Bo Li. Semi-dynamic load balancing: Efficient distributed learning in non-dedicated environments. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 431–446, 2020.

[6] Jianghao Chen, Junhong Wu, Yangyifan Xu, and Jiajun Zhang. Ladm: Long-context training data selection with attention-based dependency measurement for llms. *arXiv preprint arXiv:2503.02502*, 2025.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[8] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

[9] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115, 2023.

[10] Jiangfei Duan, Shuo Zhang, Zerui Wang, Lijuan Jiang, Wenwen Qu, Qinghao Hu, Guoteng Wang, Qizhen Weng, Hang Yan, Xingcheng Zhang, et al. Efficient training of large language models on distributed infrastructures: a survey. *arXiv preprint arXiv:2407.20018*, 2024.

[11] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407, 2024.

[12] Maxim Enis and Mark Hopkins. From llm to nmt: Advancing low-resource machine translation with claude. *arXiv preprint arXiv:2404.13813*, 2024.

[13] Fabrizio Gilardi, Meysam Alizadeh, and Maël Kubli. Chatgpt outperforms crowd workers for text-annotation tasks. *Proceedings of the National Academy of Sciences*, 120(30):e2305016120, 2023.

[14] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, et al. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793*, 2024.

[15] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 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*, ASPLOS 2023, page 266–280, New York, NY, USA, 2023. Association for Computing Machinery.

[16] Rong Gu, Yuquan Chen, Shuai Liu, Haipeng Dai, Guihai Chen, Kai Zhang, Yang Che, and Yihua Huang. Liquid: Intelligent resource estimation and network-efficient scheduling for deep learning jobs on distributed gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):2808–2820, 2021.

[17] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

[18] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 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*, pages 1–15, 2021.

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

[20] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. 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*, pages 457–472, 2023.

[21] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui

Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[22] Hugging Face, Inc. Hugging face. https://huggingface.co/, 2025. Accessed: 2025-09-22.

[23] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 721–739, 2021.

[24] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

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

[26] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[27] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

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

[29] ChonLam Lao, Minlan Yu, Aditya Akella, Jiamin Cao, Yu Guan, Pengcheng Zhang, Zhilong Zheng, Yichi Xu, Ennan Zhai, Dennis Cai, et al. Trainmover: An interruption-resilient and reliable ml training runtime. *arXiv preprint arXiv:2412.12636*, 2024.

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

[31] Haoyang Li, Fangcheng Fu, Hao Ge, Sheng Lin, Xuanyu Wang, Jiawen Niu, Yujie Wang, Hailin Zhang, Xiaonan Nie, and Bin Cui. Malleus: Straggler-resilient hybrid parallel training of large-scale models via malleable data and model parallelization. *Proceedings of the ACM on Management of Data*, 3(3):1–28, 2025.

[32] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed {MoE} training and inference with lina. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 945–959, 2023.

[33] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 835–850, 2023.

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

[35] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. *arXiv preprint arXiv:2105.13120*, 2021.

[36] Wenxue Li, Xiangzhou Liu, Yunxuan Zhang, Zihao Wang, Wei Gu, Tao Qian, Gaoxiong Zeng, Shoushou Ren, Xinyang Huang, Zhenghang Ren, et al. Revisiting rdma reliability for lossy fabrics. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 85–98, 2025.

[37] Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth*, 1(1):9, 2024.

[38] Xinyu Lian, Sam Ade Jacobs, Lev Kurilenko, Masahiro Tanaka, Stas Bekman, Olatunji Ruwase, and Minjia Zhang. Universal checkpointing: A flexible and efficient distributed checkpointing system for {Large-Scale}{DNN} training with reconfigurable parallelism. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 1519–1534, 2025.

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

[40] Jia Liu and Min Chen. Fagel: Fabric llms agent empowered embodied intelligence evolution with autonomous human-machine collaboration. *arXiv preprint arXiv:2412.20297*, 2024.

[41] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient {GPU} cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[42] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. {CheckFreq}: Frequent,{Fine-Grained}{DNN} checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216, 2021.

[43] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond {GPUs} for {DNN} scheduling on {Multi-Tenant} clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, 2022.

[44] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[45] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–15, 2021.

[46] Bogdan Nicolae, Jiali Li, Justin M Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 172–181. IEEE, 2020.

[47] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[48] NVIDIA Corporation. CUDA Runtime API :: CUDA Toolkit Documentation - Events. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html, 2025. Accessed: 2025-09-22.

[49] NVIDIA Corporation. Inter-process communication. https://developer.nvidia.com/docs/drive/drive-os/6.0.8/public/drive-os-linux-sdk/common/topics/nvsci_nvsciipc/Inter-ProcessCommunication1.html, 2025. Accessed: 2025-09-22.

[50] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[51] pybind11 Contributors. pybind11. https://github.com/pybind/pybind11, 2025. Accessed: 2025-09-22.

[52] PyTorch Contributors. Pytorch. https://pytorch.org/, 2025. Accessed: 2025-09-22.

[53] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18. USENIX Association, July 2021.

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

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

[56] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3505–3506, 2020.

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

[58] Siddharth Singh, Olatunji Ruwase, Ammar Ahmad Awan, Samyam Rajbhandari, Yuxiong He, and Abhinav Bhatele. A hybrid tensor-expert-data parallelism approach to optimize mixture-of-experts training. In *Proceedings of the 37th International Conference on Supercomputing*, pages 203–214, 2023.

[59] Honghao Song, Liang Wang, Xiaozhen Qiao, Yifan Chen, Da Sun, and Zhe Sun. Embodied intelligence for robot manipulation: development and challenges. *Vicinagearth*, 2(1):8, 2025.

[60] Andrew Szot, Max Schwarzer, Harsh Agrawal, Bogdan Mazoure, Walter Talbott, Katherine Metcalf, Natalie Mackraz, Devon Hjelm, and Alexander Toshev. Large language models as generalizable policies for embodied tasks. *arXiv preprint arXiv:2310.17722*, 2023.

[61] Qwen Team. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2, 2024.

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

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

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

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

[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[67] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. Tenplex: Dynamic parallelism for deep learning using parallelizable tensor collections. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 195–210, 2024.

[68] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, et al. {ByteCheckpoint}: A unified checkpointing system for large foundation model development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 559–578, 2025.

[69] Guanhua Wang, Olatunji Ruwase, Bing Xie, and Yuxiong He. Fastpersist: Accelerating model checkpointing in deep learning. *arXiv preprint arXiv:2406.13768*, 2024.

[70] Zheng Wang, Anna Cai, Xinfeng Xie, Zaifeng Pan, Yue Guan, Weiwei Chu, Jie Wang, Shikai Li, Jianyu Huang, Chris Cai, et al. Wlb-llm: Workload-balanced 4d parallelism for large language model training. *arXiv preprint arXiv:2503.17924*, 2025.

[71] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 364–381, 2023.

[72] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, Zhenkun Cai, Yuzhen Huang, James Cheng, Han Yuan, and Fan Yu. Elastic deep learning in multi-tenant gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 33(1):144–158, 2021.

[73] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.

[74] Lei Xie, Jidong Zhai, Baodong Wu, Yuanbo Wang, Xingcheng Zhang, Peng Sun, and Shengen Yan. Elan: Towards generic and efficient elastic training for deep learning. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 78–88. IEEE, 2020.

[75] Lingling Xu, Haoran Xie, Si-Zhao Joe Qin, Xiaohui Tao, and Fu Lee Wang. Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment. *arXiv preprint arXiv:2312.12148*, 2023.

[76] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[77] Fanlong Zeng, Wensheng Gan, Yongheng Wang, and Philip S Yu. Distributed training of large language models. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 840–847. IEEE, 2023.

[78] Tianyi Zhang, Faisal Ladhak, Esin Durmus, Percy Liang, Kathleen McKeown, and Tatsunori B Hashimoto. Benchmarking large language models for news summarization. *Transactions of the Association for Computational Linguistics*, 12:39–57, 2024.

[79] Xinyi Zhang, Hanyu Zhao, Wencong Xiao, Xianyan Jia, Fei Xu, Yong Li, Wei Lin, and Fangming Liu. Rubick: Exploiting job reconfigurability for deep learning cluster scheduling. *arXiv preprint arXiv:2408.08586*, 2024.

[80] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis CM Lau, Yuqi Wang, Yifan Xiong, et al. {HiveD}: Sharing a {GPU} cluster for deep learning with guarantees. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 515–532, 2020.

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

A Performance Model Detail

In this section, we (1) clarify *Suika*'s representation of asymmetric 3D-parallelism plan; (2) present the complete formulation of performance model, including iteration latency and peak GPU memory; (2) demonstrate how to derive coefficients from profiled observations.

Coefficients and symbols are summarized in Tab. 2 for reference.

A.1 Asymmetric 3D-parallelism Plan Representation

Suika abstracts 3D plans in PP-DP-TP hierarchy, where a plan P consists of $\overline{PP}$ stages, with the i-th stage as DP group $\mathcal{D}_i$. Inside a DP group, there are $\overline{DP}_i$ TP groups $\mathcal{T}_{i,j}$ of the same TP degree $\overline{TP}_i$. Inside a TP group, there are $\overline{TP}_i$ GPUs $\mathcal{G}_{i,j,k}$ with meta information of GPU type $(\text{type})_{i,j}$ and node id $(\text{node})_{i,j}$.

At PP level, the hyperparameter N_b denotes the number of micro-batches in a mini-batch. At DP level, l_i denotes the number of layers assigned to $\mathcal{D}_i$. At TP level, $b_{i,j}$ denotes the local micro batch size assigned to $\mathcal{T}_{i,j}$.

Due to the intense communication demand of tensor parallel, the GPUs inside a TP group are restricted to be homogeneous and on the same node. For effective gradient synchronization communication, we restrict TP degree to be the same inside a DP group for alignment. However GPU type may vary across TP groups inside a DP group, and DP / TP degree may differ across PP stages. The pipeline strategy is by default 1F1B [44].

This 3D plan representation is general and compatible with asymmetric plans defined by *Metis* [64].

A.2 Modeling Iteration Time

We model latency in a PP-DP-TP hierarchy, as illustrated in Fig. 7. We describe it with homogeneous setup for simplicity, and omit $k^{(A)}$ GPU-type superscript for certain coefficients (please refer to Tab. 2). To generalize to heterogeneous setup, simply replace coefficients with GPU-type-specific versions according to target $\mathcal{G}_{i,j,k}$.

Modeling TP. For $\mathcal{T}_{i,j}$, we aim to estimate (1) $T^{i,j}_{\text{TP-fwd}}$: the forward time; (2) $T^{i,j}_{\text{TP-bwd}}$: the backward time excluding gradient synchronization; and (3) $T^{i,j}_{\text{TP-optim}}$: the optimizer time.

$T^{i,j}_{\text{TP-fwd}}$ is decomposed into computation $T^{i,j}_{\text{TP-comp-fwd}}$ and communication $T^{i,j}_{\text{TP-commu-fwd}}$.

(i) The computation time is modeled as proportional to number of layers, local micro batch size, and the inverse of TP degree: $T^{i,j}_{\text{TP-comp-fwd}} = k_{\text{comp}} * b_{i,j} * l_i / \overline{TP}_i$, where k_{comp} is the GPU-type specific computation constant.

(ii) The communication time follows a ring all-reduce model. Under Megatron-style tensor parallelism, each transformer layer requires 2 all-reduce operations during the forward pass. The communication volume is thus $V = 2k_{\text{actv}} * b_{i,j} * l_i * 2(1 - \frac{1}{\overline{TP}_i})$. The volume per transmission $V' = V/(2l_i)$ (at order of tens to hundreds of MB) is often too small to saturate NVLINK. We empirically estimate the effective bandwidth as linear to $\log_2 V'$, up to a saturation threshold: bandwidth $= k^{(\text{node})_{i,j}}_{\text{bw-intra}} * \min(\frac{\log_2 V'}{\log_2 k^{(\text{node})_{i,j}}_{\text{bw-intra-sat}}}, 1)$, and communication time is given by $T^{i,j}_{\text{TP-fwd-commu}} = \frac{V}{\text{bandwidth}}$.

Symbol	Description
i, j, k	pipeline / data / tensor parallel indices
P	a 3D Plan in PP-DP-TP hierarchy
$\mathcal{D}_i$	DP group (PP stage i)
$\mathcal{T}_{i,j}$	TP group at PP stage i, DP replica j
$\mathcal{G}_{i,j,k}$	GPU at PP stage i, DP replica j, TP index k
$\overline{PP}$	PP degree
$\overline{DP}_i$	DP degree of PP stage i
$\overline{TP}_i$	TP degree of PP stage i
N_b	number of micro batches
l_i	number of layer for $\mathcal{D}_i$
$b_{i,j}$	local micro batch size for $\mathcal{T}_{i,j}$
$(\text{type})_{i,j}$	GPU type for $\mathcal{T}_{i,j}$
$(\text{node})_{i,j}$	node id for $\mathcal{T}_{i,j}$
$k^{(A)}_{\text{comp}}$	forward computation time per layer/sample for GPU type A
$k^{(A)}_{\text{backward}}$	backward to forward computation ratio for GPU type A
$k^{(A)}_{\text{optim}}$	optimizer time per layer for GPU type A
$k^{(A)}_{\text{overlap}}$	gradient synchronization / backward computation overlap exponent for GPU type A
$k^{(a)}_{\text{bw-intra}}$	intra-node bandwidth for node a
$k^{(a)}_{\text{bw-intra-sat}}$	saturation threshold for intra-node bandwidth for node a
$k^{(a)/(b)}_{\text{bw}}$	inter-node bandwidth between node a, b; if $a = b$, equivalent to intra-node bandwidth
k_{actv}	single activation size per sample [for TP communication]
k_{param}	(trainable) parameter/gradient size per layer [for DP communication and memory]
$k_{\text{param-optim}}$	parameter+optimizer state size per layer [for memory]
$k_{\text{actv-p}}, k_{\text{actv-np}}$	TP partitionable/non-partitionable activation size per layer/sample [for memory]

Table 2. Summarization of symbols and coefficients in performance model.

$T^{i,j}_{\text{TP-bwd}}$ is decomposed similarly. (i) Backward gradient computation time is modeled proportional to forward computation time up a constant: $T^{i,j}_{\text{TP-comp-bwd}} = k_{\text{backward}} * T^{i,j}_{\text{TP-comp-fwd}}$. (ii) The backward pass also involves exactly 2 all-reduce per transformer layer, thus, $T^{i,j}_{\text{TP-commu-bwd}} = T^{i,j}_{\text{TP-commu-fwd}}$.

$T^{i,j}_{\text{TP-optim}}$ is formulated as proportional to the parameter volume each GPU holds: $T^{i,j}_{\text{TP-optim}} = k_{\text{optim}} * l_i / \overline{TP_i}$.

Modeling DP. For $\mathcal{D}_i$, we estimate (1) the forward time as $T^{i}_{\text{DP-comp-fwd}} = \max_j T^{i,j}_{\text{TP-fwd}}$, (2) the backward time excluding gradient synchronization as $T^{i}_{\text{DP-comp-bwd}} = \max_j T^{i,j}_{\text{TP-bwd}}$, and (3) the optimizer time as $T^{i}_{\text{DP-optim}} = \max_j T^{i,j}_{\text{TP-optim}}$ where the maximum reflects potential straggler. (4) the gradient synchronization cost is $T^{i}_{\text{DP-commu}} = \frac{V}{\text{bandwidth}}$, where $V = k_{\text{param}} * l_i * 2(1 - \frac{1}{\overline{DP_i}}) / \overline{TP_i}$ [10] and bandwidth $= \min_{j_1 \neq j_2} k_{\text{bw}}^{(\text{node})_{i,j_1}/(\text{node})_{i,j_2}}$, the bottleneck bandwidth inside the DP group.

To model the overlap of gradient synchronization and backward computation, we follow existing work [53, 79] to use an exponent method: $T^{i}_{\text{DP-commu-extra}} = ((T^{i}_{\text{DP-comp-bwd}})^{k_{\text{overlap}}} + (T^{i}_{\text{DP-commu}})^{k_{\text{overlap}}})^{-1} - T^{i}_{\text{DP-comp-bwd}}$. The total extra time for last micro batch is $T^{i}_{\text{DP-extra}} = T^{i}_{\text{DP-commu-extra}} + T^{i}_{\text{DP-optim}}$.

Let $T^{i}_{\text{DP-comp}} = T^{i}_{\text{DP-comp-fwd}} + T^{i}_{\text{DP-comp-bwd}}$ be the total computation time per micro batch of the i-th stage.

Modeling PP. for P, when using 1F1B strategy, the total iteration time can break down into three parts:

$$T_{\text{PP}} = T_{\text{warm-up}} + T_{\text{steady}} + T_{\text{extra}}.$$

(1) $T_{\text{warm-up}} = \sum_i T^{i}_{\text{DP-comp}}$ represents the first micro batch at warm-up phase.

(2) $T_{\text{steady}} = (N_b - 1) * \max_i(T^{i}_{\text{DP-comp}})$ captures the remaining $N_b - 1$ micro batch at steady phase, bottle-necked by the slowest stage.

(3) $T_{\text{extra}} = \max_i(T^{i}_{\text{DP-extra}} - \sum_{j=0}^{i-1} T^{j}_{\text{DP-comp-bwd}})$ models the extra time of gradient synchronizing and optimizer step at last micro batch.

The activation communication cost between pipeline stages is generally negligible and not modeled in our formulation.

A.3 Modeling Peak Memory

It is crucial to accurately estimate peak GPU memory consumption to avoid Out-Of-Memory.

The peak memory usage on GPU $\mathcal{G}_{i,j,k}$ consists of four parts:

$$M_{\text{peak}} = M_{\text{param-optim}} + M_{\text{actv}} + M_{\text{grad}} + M_{\text{overhead}}.$$

[10] For parameter-efficient fine-tuning jobs, k_{param} should refer to only trainable parameters that have gradients

(1) $M_{\text{param-optim}}$ includes parameters and optimizer state except gradients, and is formulated as $k_{\text{param-optim}} * l_i / \overline{TP_i}$ given the repetitive nature of transformer layers.

(2) M_{actv} measures activation memory and is split into two parts: tensor-parallel partition-able portion $M_{\text{actv-p}}$, and remaining non-partition-able portion $M_{\text{actv-np}}$. This separation is essential, and neglecting it causes over 50% estimation error for $\overline{TP_i} = 1$ when using coefficients obtained from $\overline{TP_i} = 8$.

The activation memory is proportional to local micro batch size $b_{i,j}$, number of layers l_i, and number of micro batches whose activations concurrently present in memory $\min(\overline{PP} - i, B)$. Formally,

$$\begin{aligned} M_{\text{actv}} &= M_{\text{actv-p}} + M_{\text{actv-np}} \\ &= b_{i,j} * l_i * \min(\overline{PP} - i, N_b) * (k_{\text{actv-p}} / \overline{TP_i} + k_{\text{actv-np}}). \end{aligned}$$

(3) M_{grad} estimates the contribution of gradients in peak memory usage. If not all forward passes finish before first backward pass, or equivalently $N_b > \overline{PP} - i$, which is the general case, then gradients will co-exist with activation when memory peaks, so $M_{\text{grad}} = k_{\text{param}} * l_i / \overline{TP_i}$. Otherwise, $M_{\text{grad}} = 0$.

(4) M_{overhead} measures the memory not explicitly requested by user application, including CUDA context, NCCL buffer and so on. M_{overhead} is both device- and application-related, and therefore, hard to predict precisely. We choose to overclaim a small portion of memory to account for M_{overhead} and potential estimation errors of other components. The portion is empirically set to 10%, which is reasonable as the typical M_{overhead} value is around 5 GB on H100 80G.

For non-repetitive layers at the beginning or end of LLM models (e.g., embedding layer, LM classifier), the computation cost is negligible, and memory can be modeled similarly as above and added to GPUs at first/last pipeline stage.

A.4 Deriving Coefficient Solely With Online Profiled Observations

In this section, we elaborate on the derivation of all coefficients from prior knowledge and profiling data with accuracy and user-friendliness.

A.4.1 White-box Coefficients

The white-box coefficients that can be known aprior falls into two categories.

Cluster-wide Hardware Coefficients: including $k_{\text{bw}}, k_{\text{bw-intra}}, k_{\text{bw-intra-sat}}$, GPU theoretical TFLOPS and memory capacity. These can be learned from cluster hardware specification. For a more accurate effective bandwidth characterization, cluster administrator may profile them once and reuse for all jobs.

Application-implied Coefficients: including $k_{\text{actv}}, k_{\text{param}}, k_{\text{param-optim}}, k_{\text{actv-p}}, k_{\text{actv-np}}$. In theory, these constants can be determined once the application is provided, with a lot of details requiring users to report. In

practice, for a more user-friendly interface, we treat them as hidden and automatically infer them from data collected via *Suika Engine*. We will cover them in next part together with hidden coefficients.

A.4.2 Hidden Coefficients

In this section, we first introduce the engineering co-design in *Suika Engine* and profiling principles, then detail on the derivation of hidden coefficients.

Transparent profiling integration and principles. To enable elastic asymmetric 3D-parallelized training, with re-deployment optimization and online profiling, we have implemented a from-scratch *Suika*Engine. This allows us to insert intrusive profiling entry points into the framework, while keeping them transparent to users.

We implement a profiler utility which wraps the target code region to collect high-resolution GPU time with CUDA event, and memory metrics with `torch.cuda.memory_stats`. Since we do not profile as detailed as kernel-level latencies, insert only tens of CUDA events and one memory metric collection per iteration, their overheads are negligible.

Some additional metrics, such as the shape of activation tensor at pipeline stage boundary and total number of parameters, are also collected to infer aforementioned *application-implied coefficients*.

We follow two general profiling principles: (1). Only profile observable latencies, avoid those hard to track or have overlap: infer $T_{\text{TP-comp-fwd}}$ from $T_{\text{TP-fwd}} - T_{\text{TP-commu-fwd}}$ instead of profiling directly, since the latter two are easier to track; measure $T_{\text{TP-commu-fwd}}$ at forward not backward to circumvent overlap with computation. (2). Prefer profiled value over theoretical estimation whenever possible: though the $T_{\text{TP-comp-fwd}}$ can be estimated with volume / bandwidth, its profiled value is favored for coefficient derivation. These two principles are critical for stable coefficients and accurate prediction despite the various parallelization plans they may be obtained from.

Inverting the equations. First, consider homogeneous setup with only one GPU type A. Select an arbitrary $\mathcal{G}_{i,j,k}$ with its profiled data.

(1) k_{comp}: retrieved from linear relation with $T_{\text{TP-comp-fwd}} = T^{\text{(prof)}}_{\text{TP-fwd}} - T^{\text{(prof)}}_{\text{TP-commu-fwd}}$ [11].

(2) k_{backward}: $T_{\text{TP-comp-bwd}} = T^{\text{(prof)}}_{\text{TP-bwd}} - T_{\text{TP-commu-bwd}}$ where $T_{\text{TP-commu-bwd}} = T^{\text{(prof)}}_{\text{TP-commu-fwd}}$; then $k_{\text{backward}} = \dfrac{T_{\text{TP-comp-bwd}}}{T_{\text{TP-comp-fwd}}}$.

(3) k_{optim}: retrieved from linear relation with $T^{\text{(prof)}}_{\text{TP-optim}}$.

(4) k_{overlap}: notice that when $T^{\text{(infer)}}_{\text{DP-commu}}$, $T^{\text{(prof)}}_{\text{DP-comp-bwd}}$ are given, $T^{\text{(prof)}}_{\text{DP-commu-extra}}$ is monotonically decreasing with k_{overlap}. It can be numerically solved with binary search by aligning estimated $T_{\text{DP-commu-extra}}$ to profiled value. If no data-parallel is adopted and thus no profile available, pessimistically fall-back to 1 (no overlap).

(5) k_{actv}: inferred from the shape of activation (or output) at pipeline stage boundary.

(6) k_{param}: inferred from the total size of parameters that require gradients with PyTorch interface. (k_{param} must be acquired first to infer $T^{\text{(infer)}}_{\text{DP-commu}}$ mentioned earlier.)

(7) $k_{\text{param-optim}}$: inferred from the total size of model's and optimizer's `state_dict`.

(8) $k_{\text{actv-p}}/k_{\text{actv-np}}$: $k_{\text{actv-np}} = C_{\text{np}} * k_{\text{actv}}$ where C_{np} is determined by model architecture (e.g., $C_{\text{np}} = 5$ for GPT2 and Qwen2, 7 for LlaMA2). The $M^{\text{(prof)}}_{\text{observable-peak}} = M^{\text{(infer)}}_{\text{param-optim}} + M_{\text{actv}} + M^{\text{(infer)}}_{\text{grad}} (= M_{\text{peak}} - M_{\text{overhead}})$ is exactly the value from `torch.cuda.max_allocated_memory()`. Therefore M_{actv} can be inferred; by knowing $k_{\text{actv-np}}$, the last coefficient $k_{\text{actv-p}}$ can be retrieved — Moreover, if data from two GPUs with different TP degrees are available, $k_{\text{actv-p}}/k_{\text{actv-np}}$ can be jointly solved with a linear system without knowing C_{np} a prior.

Next, we discuss how to generalize the model to heterogeneous setup; Consider two GPU types A, B, and GPU-type-specific coefficients $k_{\text{comp}}, k_{\text{backward}}, k_{\text{optim}}, k_{\text{overlap}}$.

(1) If both GPU types A, B present in current plan, then two sets of coefficients are derived separately.

(2) If current plan contains only one GPU types, say A, then one-shot approximation is applied to derive $k^{(B)}$ from $k^{(A)}$. Specifically, k_{comp} is scaled by the inverse of hardware theoretical TFLOPS: $k^{(B)}_{\text{comp}} = \dfrac{\text{TFLOPS}^{(A)}}{\text{TFLOPS}^{(B)}} k^{(A)}_{\text{comp}}$; the same TLFOPS scaling applies to k_{optim}. The remaining coefficients, k_{backward} and k_{overlap}, are assumed to be identical across GPU-types when no profile data are available.

B Time Complexity Analysis For Alg. 1 Incremental Plan Solver

We list symbols for complexity analysis in Tab. 3.

Symbol	Description
N	$\lvert R^0_J \rvert$, number of GPUs in current allocation
M	$\lvert R^\Delta \rvert$, number of GPUs in additional resources
C	number of GPUs per node
B	global batch size / number of micro batches
L	number of layers in the model
S	number of pipeline stages

Table 3. Symbols in Alg. 1's Complexity Analysis

We start from low-level functions and progress to high-level algorithm.

[11]We use (prof) super-scripts to denote directly profiled values, otherwise values are by default inferred from profiled data or theoretical estimation. We may sometimes use (infer) to emphasize that the value is estimated with coefficients known so far.

(1) The throughput estimation $\mathrm{T}(P)$, including memory feasibility test, requires traversing all GPUs in the plan, leading to $O(N+M)$.

(2) The Load-Balance(P) calls a breakdown version of $\mathrm{T}(P)$ (same complexity) once to obtain lower-level latencies for reference. Load-Balance(P) models an abstract problem to split n items into m weighted bins to minimize the maximum cost; Formally, the i-bin has a cost c_i per item receives, the goal is to:

$$\min_{\{a_i\}\in\mathbb{N}^m} \max_{i\in[m]} a_i * c_i, \quad \text{s.t.} \sum_{i\in[m]} a_i = n.$$

This can be solved with heap-based greedy algorithm with complexity $O(n \log m)$. By substituting cost with low-level latency, and items as number of layers or local batch size, this algorithm can apply to both partitioning layers ($n = L$, $m = S$) and local batch sizes ($n = B$, $m = \overline{DP_i} \leq N + M$, repeated for each stage), leading to a total complexity of $O((N+M) + L \log S + SB \log(N+M))$.

(3) As for 3D-Expand-and-Balance, the PP/DP/TP-Expand respectively creates $O(\log C)$, $O(S)$, $O(S)$ candidate plans, and each candidate requires a Load-Balance and a throughput estimation call. The complexity comes to $O((\log C + S)((N+M) + L \log S + SB \log(N+M)))$

(4) In incrementalParallelize, for $\bar{P}_i$, it enumerates i predecessors, each with a 3D-Expand-and-Balance call. Similarly, getBestRsAndThps repeates incrementalParallelize for M times, requiring $O(M^2)$ calls to 3D-Expand-and-Balance.

The final complexity sums up to $O(M^2(\log C + S)((N + M) + L \log S + SB \log(N+M)))$. Leaving C, B, L, S as bounded constants, it trivially simplifies to $O(M^2(N + M + \log(N + M))) = O(M^2(N+M))$.

For three aforementioned advanced pruning techniques:

(i) "If $M >> N$, we may early-stop at $M \approx N$, as expanding a job significantly beyond its current allocation typically yields diminishing returns in utilization." — This reduce the loop in function getBestRsAndThps from $O(M)$ to $O(\min(N, M))$.

(ii) ' When M is large, it is not necessary to explore all predecessors. We may limit the window to closest W ones." — reduce the loop in function incrementalParallelize from $O(M)$ to $O(W)$.

(iii) "When N is large, it is not necessary to adjust at single-GPU granularity. We may group every w intra-node GPUs as the minimum unit." — This speed up the algrotihm by w times for the loop in function getBestRsAndThps.

Combining all these pruning techniques reduces the number of calls to 3D-Expand-and-Balance to $O(\min(N, M) * W/w)$, leading to final complexity $O(\min(N, M) * (N+M) * W/w) = O(\min(N, M) * \max(N, M) * W/w) = O(NMW/w)$ [12].