HeteroSched: Co-Optimizing Scheduling and Parallelization for Deep Learning Workloads for Heterogeneous GPU Clusters

Bahram Afsharmanesh
University of Georgia
Athens, Georgia, USA
bahram.afsharmanesh@uga.edu

Md Musfiqur Rahman Sanim
University of Georgia
Athens, Georgia, USA
musfiqur.sanim@uga.edu

Amirali Mirian
University of Georgia
Athens, Georgia, USA
AmirAli.mirian@uga.edu

Gagan Agrawal
University of Georgia
Athens, Georgia, USA
gagrawal@uga.edu

Abstract

Parallelization of individual jobs and workload scheduling in multi-tenant clusters have both been extensively explored in High Performance Computing (HPC). As Deep Learning (DL) training has emerged as a dominant workload on clusters in recent years, there has been significant interest in parallelizing individual models and scheduling them efficiently. However, to date, very few efforts have taken an integrated approach to addressing both problems, and especially have not simultaneously accounted for heterogeneity in the execution environment. We posit that adaptively parallelizing jobs based on available resources can significantly enhance overall resource utilization. This paper introduces `HeteroSched` and demonstrates that integrated optimization of parallelization and scheduling is both feasible and highly effective. The main insight of our work is to generate a set of possible target resources for each job and to develop corresponding parallelization strategies for each. Each job-strategy pair is then treated as a scheduling candidate. Scheduling decisions are made by extending prior Linear Programming (LP)-based formulations and incorporating new heuristics to reduce decision overhead. Our evaluation against three baselines demonstrates improvements of 40% to 3× in makespan and average job completion time.

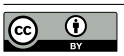
CCS Concepts

• **Computer systems organization**; • **Computing methodologies** → **Machine learning**; **Parallel computing methodologies**;

Keywords

Deep Learning Training, Heterogeneous GPU Clusters, Cluster scheduling

ACM Reference Format:
Bahram Afsharmanesh, Md Musfiqur Rahman Sanim, Amirali Mirian, and Gagan Agrawal. 2026. HeteroSched: Co-Optimizing Scheduling and Parallelization for Deep Learning Workloads for Heterogeneous GPU Clusters. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '26), October 19-22, 2026, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3838684.3846882



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846882

1 Introduction

The rapid emergence of deep learning (DL) models with ever-increasing sizes has made the training of such models a dominant HPC workload. Due to their large computational demands, such training is conducted in a distributed fashion on clusters equipped with heterogeneous hardware accelerators such as GPUs, TPUs, and FPGAs [5, 12]. While these clusters provide substantial computational power and flexibility, they also introduce significant challenges in scheduling and parallelization, particularly due to variations in processing speed, memory capacity, and interconnect bandwidth within a large cluster [24, 25]. The emergence of such workloads and the adoption of heterogeneous clusters have led to extensive research efforts in two major areas:

Heterogeneous Scheduling for DL Workloads: The emergence of DL workloads and increased heterogeneity has led to increased interest in the classical cluster scheduling problem. Over recent years, various scheduling systems have been proposed, incorporating heterogeneity awareness [22, 31], adaptive parallelism [26], and software-hardware co-design [15] to optimize resource allocation. Most existing approaches typically focus on a single aspect of the problem—such as automatic heterogeneity-aware parallelization [33], heterogeneity-aware scheduling [22, 31], or heterogeneity-agnostic adaptive parallelization scheduling [6, 27]—while balancing the trade-off between optimization overhead and scheduling efficiency.

Parallelization Approaches for DL Training (and Inference): A DL task is composed of (and programmed with) a common operator set (matmul, conv, attention, etc.). This allows for automatic parallelization of the models. A number of approaches have been proposed—*data parallel* [2], *tensor parallelism* [23, 29], *pipeline parallelism* [8, 17, 21], and others [2, 14]. In all, it is challenging to select the optimal parallelization scheme (or combination of schemes) given the available (heterogeneous set of) resources for execution.

While both parallelization and scheduling have been extensively studied independently, only a small number of recent systems jointly optimize them, and support for heterogeneous mixed-GPU execution remains limited. Specifically, Arena [36] co-designs dynamic scheduling and adaptive parallelism for large-model training, but only schedules homogeneous resource bundles. Sia [9] similarly

supports adaptive parallelization strategies at the scheduler level, but it also allocates homogeneous resources to each job and therefore faces the same limitation. The classical concept of malleable scheduling has also been applied to DL workloads [26] to enable elastic scaling across a cluster, but these approaches support only a fixed parallelization strategy. Gavel [22] effectively accounts for hardware and software heterogeneity in deep learning workloads; however, it lacks job-to-resource adaptivity because it does not leverage the dynamic parallelization capabilities of DL tasks.

To address this gap, we present `HeteroSched`, a system that holistically optimizes both scheduling and parallelization for deep learning (DL) workloads on heterogeneous GPU clusters. `HeteroSched` integrates adaptivity, lightweight profiling, and advanced parallelization strategies within a unified, low-overhead framework. It introduces a hierarchical two-stage optimization process: first, dynamically generating high-quality candidate parallelization strategies for each job, tailored to available heterogeneous resources; and second, selecting and scheduling the most effective job-strategy pairs to optimize scheduling objectives. Through comprehensive evaluations using trace-derived simulations, structured trace replay, and physical-cluster validation, we show that `HeteroSched` delivers substantial improvements over state-of-the-art schedulers. Compared to Gavel, it achieves up to a 3× reduction in average job completion time, and delivers a 40% improvement in makespan relative to approaches similar to Sia.

Overall, this paper makes three key contributions.

- It identifies a critical limitation in current DL schedulers: existing systems either fix job parallelization, restrict per-job allocations to homogeneous GPU types, or ignore resource heterogeneity, leading to poor adaptability and suboptimal resource utilization.
- It proposes a novel three-component architecture that decouples complexity while enabling integrated optimization: an adaptive parallelization module generates cost-aware strategies using lightweight profiling; a strategy allocator formulates and solves a parallelization-aware LP considering throughput and scheduling objectives; and a scheduling mechanism greedily maps job-strategy pairs to available resources, with support for preemption. The allocator extends Gavel's LP formulation by adding a parallelization dimension and employs a custom chunked-LP solver to improve scalability.
- It demonstrates that this integrated, low-overhead design enables flexible optimization for diverse objectives such as throughput, makespan, and average job completion time. `HeteroSched` consistently outperforms strong baselines on static workloads and Gavel under continuous arrivals. To the best of our knowledge, it is the first system to integrate auto-parallelization with heterogeneity-aware scheduling while allowing a single job to use mixed GPU types.

2 Additional Background

2.1 Deep Learning Parallelism

Deep learning parallelization refers to the process of partitioning tensors—whether they represent operator input data or model weights—and/or the computation graph of a deep learning model. This process results in three primary forms of parallelism: *data parallelism* (DP), *tensor parallelism* (TP), and *pipeline parallelism* (PP) [33], as well as hybrid schemes.

Data parallelism. DP [2] allows multiple GPUs to train model replicas in parallel, each processing a distinct data subset. After local computations, typically an all-reduce operation synchronizes gradients from the forward and backward passes, ensuring consistent model parameters. DP reduces per-GPU computation while making synchronization overhead a larger fraction of iteration time [1, 30]. In addition, standard DP does not reduce replicated model-state memory per GPU, limiting support for very large models.

Tensor parallelism. TP [23] involves partitioning model weights and associated computations into non-overlapping tensor chunks distributed across multiple GPUs. Each GPU processes a specific chunk and synchronizes with others to merge outputs, ensuring consistency in the forward and backward passes. While TP reduces memory and computational load per GPU, it incurs higher communication overhead.

Pipeline parallelism. PP [8, 21] partitions a model into sequential stages, with each stage consisting of a series of operations executed on distinct device groups. During training, activations and gradients are transferred between adjacent stages. To enhance efficiency, PP employs micro-batching, where the input batch is divided into smaller units, allowing computation on one micro-batch to overlap with communication for others. This approach enables the training of large models that exceed single-GPU memory limits. However, PP presents a trade-off: while increasing the number of stages reduces memory pressure per stage, it also raises communication overhead due to increased inter-stage data transfers.

Hybrid parallelism. Recent advancements indicate that the aforementioned individual (*one-dimensional or 1D*) parallelism strategies must be integrated to effectively scale modern large-scale models [23, 35]. Various parallelism combinations exist [13, 23, 34, 35], but a common classification defines 2D parallelism as the combination of DP and TP, while 3D parallelism extends 2D by incorporating PP [19, 28, 33].

2.2 Heterogeneity-Aware and Adaptive DL Scheduling

Modern GPU clusters can exhibit substantial heterogeneity across compute capabilities, memory bandwidth, and interconnect performance. DL workloads have also become increasingly diverse. Some models, such as Wide-ResNet [37], exhibit layerwise heterogeneity, with different layers having varying computational and memory demands, while others, even if architecturally homogeneous across layers, exhibit significant performance variation across GPU types due to hardware-specific bottlenecks and scaling inefficiencies [22].

Recent cluster schedulers have begun to incorporate awareness of both hardware and model heterogeneity when managing deep learning (DL) workloads. Gavel [22], also mentioned previously, accounts for hardware heterogeneity by leveraging throughput profiles across device types to guide job placement. More general approaches explore *moldable* and *malleable* scheduling paradigms. *Moldable* jobs permit the selection of the number of computing units (e.g., GPUs) prior to execution, while *malleable* jobs support dynamic reconfiguration at runtime [3]. Fujiwara et al. [4] show that even simple dynamic adjustment schemes can significantly

improve average job completion time using measured data-parallel-style speedup curves, but do not model tensor- or pipeline-parallel tradeoffs. ElasticFlow [6] incorporates adaptivity into a serverless scheduler, guaranteeing minimum resource allocations to meet job deadlines and greedily assigning surplus resources based on marginal throughput gains.

3 Integrated Framework

3.1 Problem Definition and System Overview

As a background, a deep learning model is represented as a computational graph with L layers (see Figure 2a). A *parallelization strategy* partitions this graph into pipeline stages (pp) and assigns data parallelism degree (dp) and tensor parallelism degrees (tp) to each stage, which are then mapped to a set of GPUs. For example, Job 1 in Figure 2 is mapped to a (2,2) subset—two GPUs from each of two nodes with different GPU types. `HeteroSched` uses round-based scheduling to allocate jobs and GPU resources, as Figure 2b shows. In each round, it selects jobs from the queue based on cluster objectives (e.g., maximizing throughput) and assigns a parallelization strategy per job. For instance, for Job 1, the scheduler selects the (2,2) subset with pp=2, stage 1 configured as ($dp = 2$, $tp = 1$), and stage 2 as ($dp = 1$, $tp = 2$) as illustrated in Figure 2a. Stage 1 (layers 1–2) is replicated on Node 1 with data parallelism, while stage 2 (layers 3–4) is tensor-parallelized across GPUs on Node 2.

`HeteroSched` separates model-specific planning from scheduling. For each (job, GPU-subset) pair, the adaptive parallelization module uses the job's profile to generate candidate strategy tuples containing resource demand and predicted throughput. Here, a job denotes a submitted training specification, including the model, global batch size, and other execution-relevant hyperparameters. A GPU-subset denotes a candidate resource bundle for a job, represented by the number of GPUs of each GPU-type subset. In each round, the *strategy allocator* assigns time fractions to job-strategy pairs under the scheduling objective, and the greedy *scheduling mechanism* instantiates the highest-priority feasible pairs on available GPUs. This hierarchy keeps the scheduler model-agnostic while still enabling optimized strategy selection over heterogeneous resources.

3.2 Adaptive Parallelization

`HeteroSched` decouples scheduling from parallelization on heterogeneous GPU clusters through a planner-independent interface. For each (job, GPU-subset) pair, the adaptive parallelization module returns a small set of candidate strategies with predicted throughput and required GPU-type counts. The scheduler uses only these tuples for LP allocation, chunked decomposition, and greedy placement. Our prototype uses Metis [33] because it efficiently searches hybrid DP/TP/PP strategies on heterogeneous GPU subsets; alternative planners or manually curated strategies can populate the same interface without changing scheduler logic. This interface is vendor-neutral: supporting other vendors requires vendor-specific profiling and execution backends, while the scheduling optimization remains unchanged. We formalize the strategy tuple in §3.3.

Upon job arrival, `HeteroSched` initiates a lightweight, two-phase profiling pass integrated within the *job strategy planner* that was shown earlier in Figure 1. In the first phase, the model is statically traced using PyTorch's *meta device*, enabling per-layer measurements. The measured values include parameter counts, tensor shapes, and estimated activation sizes—these values are obtained without any memory allocation or execution. In the second phase, the system obtains runtime measurements through a short execution. Specifically, the system determines whether the model fits within a single GPU; if not, it automatically applies minimal model parallelism to partition the computational graph across the smallest viable set of homogeneous GPUs. It then launches a short profiling run under realistic execution conditions, measuring forward and backward latencies per layer, memory consumption for activations and weights, and optimizer-related overheads.

After profiling completes, `HeteroSched` enumerates a constrained set of heterogeneous GPU subsets to limit the search space and improve tractability. Specifically, (1) subsets are restricted to powers-of-two in GPU count to simplify mapping and increase the likelihood of schedulable configurations, and (2) the aggregate memory capacity of each candidate subset must be no more than 2× the job's minimum memory requirement. Each (job, GPU-subset) pair is passed to Metis, which explores the space of pipeline-parallel configurations by grouping layers into stages and determining stage-wise degrees of tensor and data parallelism. The objective is to minimize total execution cost over feasible parallelization plans:

$$T^* = \min\{\sum_{m=1}^{S} t_m + (B-1) \cdot \max_{1 \le n \le S} t_n + DP_{all}\},$$

where S is the number of pipeline stages, B is the number of micro-batches, t_m is the latency of stage m, and DP_{all} captures all-reduce communication overhead. The output includes a strategy which denotes a fully specified parallelization plan for a particular (job, GPU-subset) pair, including: pipeline degree pp, stage-wise tensor- and data-parallel degrees tp and dp, micro-batch size mb, and estimated execution cost T^*. All profiling results and corresponding module outputs are cached, enabling fast reuse of previously derived strategies.

3.3 Scheduler

Strategy Allocator: To determine how GPU resources should be allocated across jobs and their corresponding parallelization strategies, `HeteroSched` models the scheduling problem as a global linear programming (LP) problem. This formulation significantly extends prior work [7, 16, 22] by explicitly supporting parallelization-aware scheduling. Here, each job is associated with a set of pre-evaluated strategies, each characterized by unique resource demands and estimated throughput. More specifically, we extend the decision variable with an additional dimension to capture parallelization strategies, enabling a more expressive and adaptive formulation. Furthermore, we relax both the strategy allocation and scheduling mechanisms to allow flexible alignment with the scheduling objective, as long as these choices remain non-conflicting. This co-design enables `HeteroSched` to better exploit the heterogeneity and dynamic resource availability of modern GPU clusters.

Let $\mathcal{J}$ denote the set of active jobs and let S_i denote the candidate strategy set for job i. Each strategy $j \in S_i$ is represented by its heterogeneous GPU demand vector $(r_{ij}^1, r_{ij}^2, \ldots)$ and its predicted

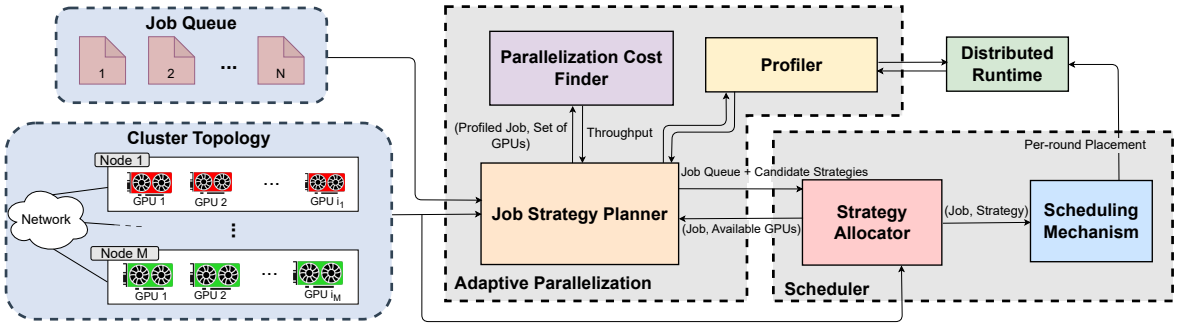
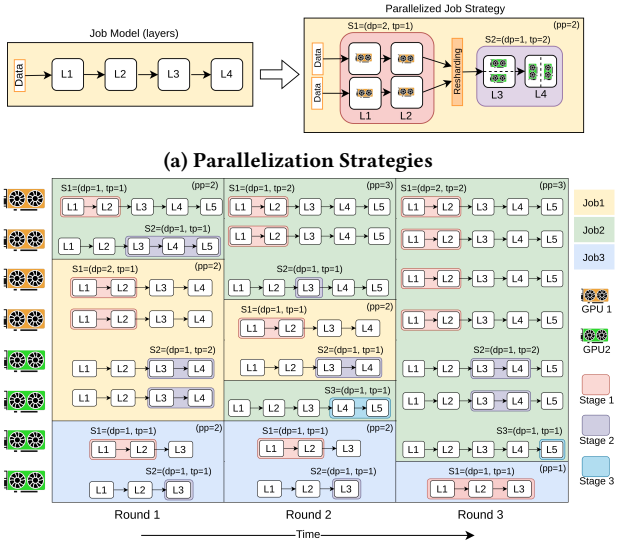


Figure 1: Heterogeneous GPU Cluster Parallelization and Scheduling Overview



(a) Parallelization Strategies

(b) Integrated Scheduling and Parallelization

Figure 2: Example of integrated scheduling and parallelization. (a) Job 1 uses two pipeline stages with $(dp, tp) = (2, 1)$ and $(1, 2)$, respectively. (b) Jobs receive GPU allocations and parallelization strategies at each scheduling round.

throughput $\text{thpt}_{i,j}$, where r_{ij}^k denotes the number of GPUs of type k allocated to job i under strategy j. Let W be the set of GPU types and C^k be the available number of GPUs of type k. The LP decision variable $x_{i,j}$ denotes the target fraction of the scheduling horizon assigned to strategy j of job i, and $X = \{x_{i,j} \mid i \in \mathcal{J}, j \in S_i\}$ denotes the full allocation vector. The LP computes a globally consistent allocation over all $(job, strategy)$ pairs. Our formulation supports multiple objectives, including aggregate-throughput maximization and makespan minimization, subject to the following constraints.

Feasibility constraint. Each strategy's time allocation must be non-negative and cannot exceed the scheduling horizon:

$$0 \le x_{i,j} \le 1 \quad \forall i \in \mathcal{J},\ j \in S_i. \tag{1}$$

Job Over-Allocation. The total fraction of time allocated to strategies of a single job cannot exceed 1:

$$\sum_{j \in S_i} x_{i,j} \le 1 \quad \forall i \in \mathcal{J}. \tag{2}$$

Cluster Capacity. The total use of each GPU type k by all jobs and their chosen strategies must not surpass the number of available workers:

$$\sum_{i \in \mathcal{J}} \sum_{j \in S_i} x_{i,j} \cdot r_{ij}^k \le C^k \quad \forall k \in W. \tag{3}$$

Under these constraints, the objective function is expressed as: Minimize $F_{\text{policy}}(X)$, where F_{policy} is the linear function capturing the chosen scheduling policy. This LP can be solved efficiently by standard solvers, and additional policy-specific constraints can be integrated as needed. Many scheduling objectives can be realized via a single LP or a short sequence of LPs. Although the LP allows fractional allocations across multiple strategies of the same job, at runtime we commit to at most one strategy per job per round; the LP is re-solved across rounds, so all jobs with non-zero $x_{i,j}$ eventually receive time.

Runtime Adaptivity Model: In contrast to deployments that bind each job to a predefined parallelization plan, `HeteroSched` represents each job by a discrete set of candidate strategies. Adaptivity occurs only at round boundaries: at each rescheduling point, the system chooses (1) how many GPUs—and which GPU types—to allocate and (2) which pre-enumerated strategy to run; the chosen strategy remains fixed for the duration of the round. This design matches the round-based execution model used by our greedy placement mechanism. We assume that reconfiguration is learning-outcome neutral: strategy changes occur only at completed optimizer-step/checkpoint boundaries and preserve the user-specified global batch size through micro-batching or gradient accumulation. Changes in GPU topology may alter floating-point kernels and reduction order; therefore, we assume statistical convergence equivalence rather than bitwise reproducibility.

Scheduling Objective: We illustrate here how to transform common scheduling policies into suitable objective functions $F_{\text{policy}}(X)$. We highlight three primary objectives, which are maximizing total throughput, minimizing makespan, and minimizing average job completion time (JCT). Other policies (e.g., FIFO, Shortest-Job-First, Max-Min Fairness) can be similarly expressed in this LP framework.

Maximize Throughput. To maximize cluster-wide throughput, one seeks to choose strategy allocations that collectively maximize the total effective throughput:

$$F_{\text{MaxThpt}}(X) = -\sum_{i\in\mathcal{J}}\sum_{j\in S_i} x_{i,j} \cdot \text{thpt}_{i,j}, \tag{4}$$

here $\text{thpt}_{i,j}$ is measured in iterations/sec (i.e., training batches/sec), and the negative sign arises from the convention of converting the original *maximize* problem into a standard *minimize* form.

Minimize Makespan. Here, we target the shortest possible time T in which to complete all jobs under the above constraints. Since directly encoding makespan as a single LP objective can complicate the problem, we adopt a binary search on T. Let iter_i denote the total number of training iterations required by job i. For each candidate makespan T, we impose:

$$\sum_{j\in S_i}\left(\text{thpt}_{i,j} \cdot x_{i,j}\right) \geq \frac{\text{iter}_i}{T} \quad \forall\, i \in \mathcal{J}. \tag{5}$$

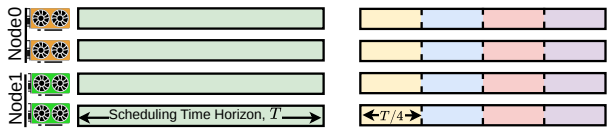
By checking feasibility (i.e., verifying whether there exists an X satisfying all constraints), we employ binary search to find the minimal feasible T. The objective can be a constant—e.g., minimize 0—since only feasibility matters for each trial T.

Minimize Average Job Completion Time. To minimize the average JCT, the scheduling objective prioritizes jobs with shorter execution times. This policy inherently risks potential starvation for longer-running jobs. To implement this prioritization while mitigating starvation, we introduce a weighting factor, w_i, representing the raw queuing wait time accumulated by job i (updated each round). Within this formulation, the decision variable $x_{i,j}$ influences the objective function to favor job-strategy pairs that contribute most effectively to reducing the average JCT, weighted by their waiting time. The objective function to minimize average JCT, subject to the constraints defined earlier, is formulated as:

$$F_{\text{MinAveJCT}}(X) = -\sum_{i\in\mathcal{J}}\sum_{j\in S_i} \frac{\text{thpt}_{i,j}}{\text{iter}_i} \cdot x_{i,j} \cdot w_i. \tag{6}$$

Here, $\text{thpt}_{i,j}/\text{iter}_i$ represents the inverse of the estimated completion time for job i using strategy j. Multiplying by the allocation fraction $x_{i,j}$ and the waiting time w_i prioritizes jobs that have waited longer and can make rapid progress.

Scalable Strategy Allocator: While the LP-based formulation described above yields globally optimal strategy assignments, solving a single, monolithic LP becomes intractable as the number of jobs and candidate strategies increases. To maintain responsiveness under high cluster load and frequent job arrivals, `HeteroSched` employs a scalable LP solving approach based on chunked workload decomposition. As shown in Figure 3, the key insight is that LP solvers inherently produce balanced allocations across available resources. This property allows the global scheduling problem to be decomposed into smaller, independent subproblems without significantly degrading solution quality. Specifically, `HeteroSched` partitions the active job queue into disjoint chunks, each containing a fixed number of jobs (e.g., 16 or 32). For each chunk $\mathcal{J}_c$, an LP is constructed using only the local job subset and its associated candidate strategies $\mathcal{S}_i$, while operating under a localized view of available resource budgets. We also empirically find that chunked LP closely preserves global-LP solution quality while substantially reducing solver overhead; see § 4.5.



(a) Non-scalable LP solving **(b) Chunked LP solving**

Figure 3: Schematic comparison of global and chunked LP allocation. (a) One LP allocates resources across all jobs. (b) Smaller LPs solve disjoint job subsets in parallel, and their allocations are merged.

Each chunk-level LP reuses the structure of the global problem but limits the problem size, ensuring that per-solver compute overhead remains bounded. Once all chunk-level LPs are solved, their outputs are merged to produce a globally consistent allocation. To maximize throughput, `HeteroSched` invokes LP solvers in parallel using a thread pool, with each chunk assigned to a separate solver instance. Each LP solve completes quickly (typically under 100ms), and the total allocation latency scales linearly with the number of chunks—ensuring predictable and efficient scheduling even under large-scale workloads.

Scheduling Mechanism: Once the candidate strategies are selected by the strategy allocator, `HeteroSched`'s Scheduling Mechanism determines which jobs to launch and where to place them across the cluster using a round-based, greedy placement algorithm. Unlike prior approaches that operate at the job level, `HeteroSched` performs scheduling at the granularity of individual (job, strategy) pairs, enabling fine-grained control and better utilization of heterogeneous resources. To guide placement decisions, the scheduler constructs a priority-sorted list of all eligible (job, strategy) pairs. Each entry is ranked according to a two-dimensional score that combines the LP-derived allocation weight and a strategy's throughput efficiency. Let $T_{i,j}$ denote the realized fraction of scheduling time already given to strategy j of job i.

$$\text{priority}_{i,j} = \left(\frac{x_{i,j}}{T_{i,j}}, \frac{\text{thpt}_{i,j}}{\sum_k r_{ij}^k}\right).$$

The ratio $x_{i,j}/T_{i,j}$ measures how far the realized fraction is from the LP target, prioritizing under-served $(job, strategy)$ pairs and steering placement toward the global plan. The second component, throughput per GPU, breaks ties within a round toward resource-efficient strategies. Together, they let the scheduler make fast local decisions. During each scheduling round, the algorithm iterates over this priority queue in descending order. For each (job, strategy) pair, it checks whether the required subset of GPUs is currently available. If placement is feasible, the job is launched with the selected strategy, the corresponding GPUs are marked as occupied, and the job is flagged as scheduled—thereby skipping all other strategies for that job. This process continues greedily until no additional strategies can be accommodated in the current round. Between rounds, priorities are recomputed to reflect updated cluster states and allocation contexts—jobs skipped earlier gain priority, reducing the risk of starvation.

4 Evaluation

4.1 Experimental Study Design

Baselines and Evaluation Objectives. Our work considers three major baseline families. Across these comparisons, we focus on makespan, average job completion time, and tail waiting time and starvation under the average-JCT objective. First, we use Gavel [22] as a strong heterogeneity-aware scheduler for deep learning workloads, but one that assumes a fixed parallelization strategy. Second, we consider *fixed-parallelism-family scheduling*, where each job is restricted to one parallelism family (DP+PP, TP+PP, or PP), while the scheduler may vary its resource allocation. We additionally evaluate a decoupled (post-hoc Metis) baseline: the scheduler first allocates GPUs using the fixed-family performance model, after which Metis selects the best parallelization plan executable on exactly that allocation, without revising the scheduler's resource decision. Third, we compare against recent integrated scheduling-and-parallelization systems. This includes Sia [9] which jointly considers scheduling and adaptivity, but only over homogeneous GPU groups and limited forms of parallelism and Arena [36], which co-designs dynamic scheduling and adaptive parallelism, but it schedules homogeneous resource bundles, rather than heterogeneous mixed-GPU subsets within a job.

Methodology. We evaluate `HeteroSched` primarily using a round-based, trace-driven GPU-cluster simulation. The simulated cluster has three nodes totaling 12, 24, or 48 GPUs drawn from A100, A6000, and RTX 4090, to reflect realistic heterogeneity [33]; we also include a homogeneous A100-only setting for comparison. `HeteroSched` retains up to the top 20 candidate strategies per job, ranked by throughput per allocated GPU. For each assigned (job, strategy) pair, the simulator computes per-round progress using the strategy throughput predicted by Metis from the profiled cost model. In every 5-minute round, jobs accrue progress equal to (effective round time) × (throughput of their assigned (job, strategy)); we model a 10-second load/checkpoint overhead per round (3.3%), and enforce memory/placement feasibility when assigning strategies to devices. Our simulation adopts the same round-based, discrete-time methodology used by prior DL schedulers that have been validated against real clusters, such as Gavel [22] and Hadar [31], which report simulation-to-real deviations of less than 8% and 10%, respectively. We extend this setup in one way: the scheduler optimizes over job–strategy pairs rather than fixed-shape jobs, while the underlying progress model remains unchanged—discrete rounds, profiled throughput from short profiling runs, and explicit checkpoint/load cost. We validate both the simulator and strategy cost model in §4.6 using an 8×A6000 homogeneous cluster and a three-node, two-GPU-type cluster with 4×L40S and 4×V100 GPUs. The latter directly evaluates predicted versus measured performance for heterogeneous parallelization strategies.

Workloads. Workloads are generated following the methodology presented in Hadar [31], using the Microsoft trace to construct representative job submissions with diverse resource and parallelization demands. A summary of the characteristics of the workload is provided in Table 1. We sample jobs from the Microsoft Philly trace [10], which provides metadata such as submission times and job durations. As the trace lacks model identity, we bucket jobs into five size classes—Small to XXLarge—by total GPU time and uniformly assign each job one of the models listed for its class in Table 1. We evaluate `HeteroSched` under two job arrival patterns: *static* and *continuous*. In the static setting, all jobs are available at the beginning of the trace. In the continuous setting, jobs arrive over time following a Poisson process with a tunable arrival rate λ (jobs/hour). To assess performance across increasing load, including overloaded conditions, we vary λ and measure the average JCT.

Overheads. We separate one-time pre-scheduling costs from runtime scheduling costs. Pre-scheduling includes profiling and parallelization planning. Profiling is performed once for a previously unseen job, while planning is performed once per previously unseen (job, GPU-subset) pair; both results are cached for reuse. Runtime overhead includes the fixed checkpoint/load cost incurred at scheduling-round boundaries and the scheduler compute time for LP solving and priority updates. In the simulator, all schedulers use the same 5-minute rounds and the same 10-second checkpoint/load cost per round. §4.5 separately evaluates sensitivity to both parameters, scheduler scalability, and the optimality gap of `HeteroSched`'s hierarchical decisions.

Table 1: Workload Properties. The Mixed workload aggregates jobs across all size categories (S, M, L, XL, XXL).

Size	Model Group	GPU-Hours
S	MoE 380M, L2 271M, WRN 250M	0–1
M	MoE 1.3B, L2 1B, WRN 1B	1–10
L	MoE 2.4B, WRN 2B, WRN 4B	10–50
XL	L2 7B, MoE 10B, WRN 6.8B	50–100
XXL	L2 13B, WRN 13B, L2 26B	100–200
Mixed	Aggregated from all S–XXL	0–200

Abbrev.: L2 = Llama 2, WRN = Wide-ResNet, MoE = Mixture-of-Experts.

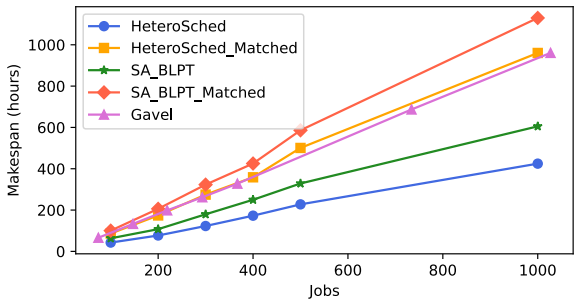
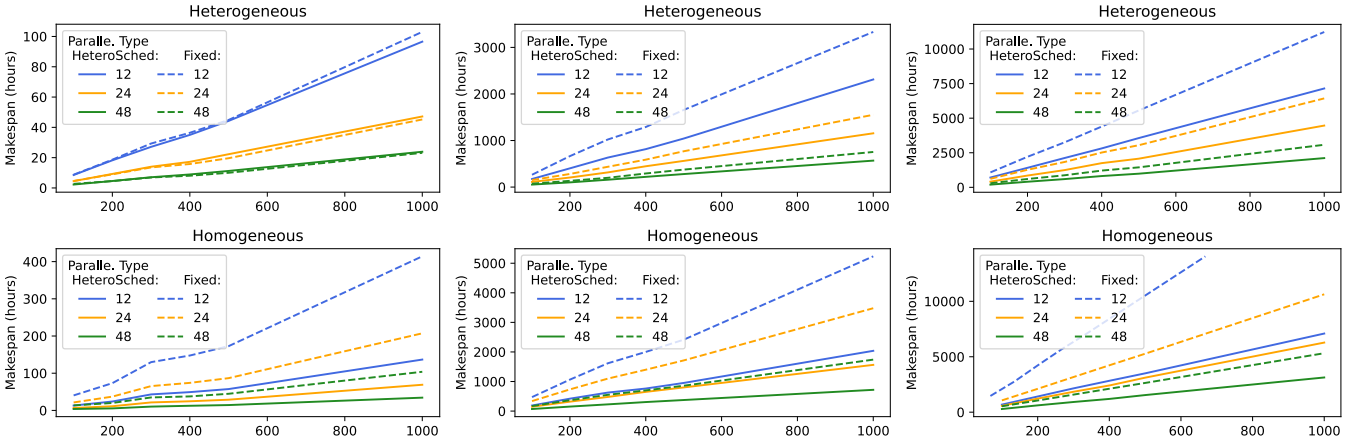


Figure 4: Makespan ablation on the 24-GPU heterogeneous Mixed workload. Matched variants use Gavel's fixed per-job plans; `SA_BLPT` uses `HeteroSched`'s adaptive strategy space with a greedy allocator replacing the LP.

4.2 Parallelization and Scheduling Gains

We first decompose `HeteroSched`'s static makespan gains, then test whether its dynamic-arrival gains persist under wait-time aging and trace-correlated burst structure. We begin with a *fixed parallelization scheme*, where the scheduler receives jobs with predetermined parallelization strategies and target GPU counts. As



(a) Workload Size: Small (b) Workload Size: Mixed (c) Workload Size: XXLarge

Figure 5: Makespan of HeteroSched (up to 20 strategies per job) versus fixed per-job parallelization on 12-, 24-, and 48-GPU clusters. Columns: (a) Small, (b) Mixed, and (c) XXLarge workloads. Rows: heterogeneous (top) and homogeneous (bottom) clusters.

discussed earlier, Gavel [22] is the state-of-the-art representative of this setting. We therefore evaluate on the Mixed workload (Table 1) using a 24-GPU heterogeneous cluster, and configure Gavel simulator with the best fixed per-job plans produced by Metis that it can execute. To separate gains from the adaptive strategy space and from the scheduler itself, we also compare against a scheme we refer to as SA_BLPT (Strategy-Aware Bottleneck-Longest-Processing-Time first), which uses the same Metis-generated adaptive strategy space as HeteroSched but replaces the LP-based allocator with a simple heterogeneity-aware greedy heuristic. We further include HeteroSched_Matched and SA_BLPT_Matched, which are restricted to the same fixed per-job plans used by Gavel.

Figure 4 shows three clear trends. First, HeteroSched_Matched remains close to Gavel when both are restricted to the same fixed plans, indicating that scheduler changes alone do not explain the large gains. Second, SA_BLPT outperforms Gavel across all workload sizes; it reduces makespan by 32.8% on average, showing that a portion of the benefit comes directly from the richer adaptive parallelization space. Third, full HeteroSched still outperforms SA_BLPT by 30.7% on average, demonstrating that the LP-based scheduler provides an additional and non-trivial benefit even when given the same adaptive strategy space. Overall, these results show that HeteroSched's improvements come from both sources: adaptive parallelization expands the set of useful job-resource mappings, and the LP scheduler more effectively coordinates those mappings across competing jobs.

Figure 5 compares HeteroSched with a fixed-strategy—best-known static parallelization configuration—baseline across homogeneous and heterogeneous clusters, workload sizes (Small, Mixed, XXL), and 12/24/48 GPUs. HeteroSched generally reduces makespan, with the smallest gains on small heterogeneous workloads where feasible strategy choices are more constrained. The benefit grows with workload size because larger queues expose more opportunities to exploit adaptive parallelization.

Next, Figure 6 illustrates the job completion time (JCT) under varying job arrival rates, comparing HeteroSched to Gavel. For each job arrival rate (λ) value, we run the experiment with three different random seeds and report the mean and standard deviation of JCT. To reduce bias from transient effects at the beginning or end of the trace, we compute JCT only for jobs with IDs ranging from 300 to 700. New jobs continue to arrive until all jobs in this target range have been completed. At higher input rates ($\lambda = 3, 4, 5$ jobs/hour), Gavel cannot sustain the offered load and experiences rapidly increasing queuing delay. In contrast, HeteroSched maintains substantially lower JCT under these high-pressure workloads, although the highest arrival rate ($\lambda = 5$) exceeds steady-state cluster capacity. For $\lambda = 2$, the figure further presents the cumulative distribution function (CDF) of the JCTs, which captures the latency behavior of the jobs with continuous arrivals. HeteroSched outperforms Gavel across the entire distribution, completing a significantly larger fraction of jobs earlier. This is reflected in a steeper and more left-shifted CDF curve, indicating both lower average and tail JCTs. Overall, we can see that compared to its performance under static workloads, HeteroSched shows even greater efficiency under dynamic and high-pressure scenarios. This improvement is driven by its ability to adaptively select parallelization strategies and flexibly match jobs to available heterogeneous GPU configurations.

Equation 6 ages each job's shortest-job-first score by its queueing time w_i, increasing its priority as it waits. We repeat the Figure 6 experiment with *aging* disabled ($w_i = 1$), while holding the workload, strategy space, allocator, and random seeds fixed (Table 2). The only methodological difference is termination: Figure 6 runs until all measured jobs complete, whereas Table 2 uses a predeclared 30-day observation horizon after the last measured job arrives so starvation can be quantified; arrivals continue until the horizon, and any measured job still unfinished is counted as starved. At $\lambda = 2$, both variants complete all jobs; aging reduces maximum

wait from 20.00 to 12.28 hours (39%) while modestly increasing S/M/L JCT. Under overloaded $\lambda = 5$, aging reduces unfinished jobs from 29.7 to 3.0, lowers p99 wait from 807 to 647 hours, and raises XL completion fraction from 53.9% to 95.3%. The lower XL JCT without aging (41.68 vs. 194.24 hours) is a truncation artifact because JCT averages completed jobs only. Thus, aging trades some short-job latency for substantially better protection of long jobs under contention.

To test sensitivity to the IID (independent and identically distributed) assumptions above, we construct three Philly replay windows that preserve inferred user burst structure. A burst consists of consecutive submissions from the same user with the same trace GPU count, separated by at most 300 s and spanning at most one hour. Jobs within a burst share a synthetic model and fixed total GPU count; at that count, `HeteroSched` may still choose the GPU-type mix and strategy. Jobs mapped to at least 50 GPU-hours are conservatively treated as fixed-plan jobs and cannot be reparallelized. The structured replay preserves intra-burst offsets and rescales inter-burst gaps to 2 jobs/hour. A matched-IID control retains the same jobs and constraints but replaces correlated arrivals with Poisson arrivals, while the published-IID control uses the independent model-assignment and Poisson-arrival methodology above. All cases use the Mixed workload, the 24-GPU heterogeneous cluster, and the same three seeds as Figure 6.

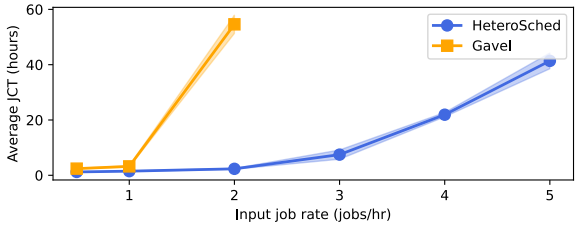
Figure 7 shows that the gain persists under correlated workload structure. Mean JCT decreases from 334.0 to 133.4 hours (2.50×) under structured replay, from 268.2 to 92.6 hours (2.89×) under matched IID, and from 243.0 to 91.2 hours (2.67×) under published IID. Preserving burst structure therefore narrows the gain relative to the matched-IID control, but `HeteroSched` still maintains a substantial reduction in mean JCT.

4.3 Fixed-Parallelism-Family and Decoupled Scheduling

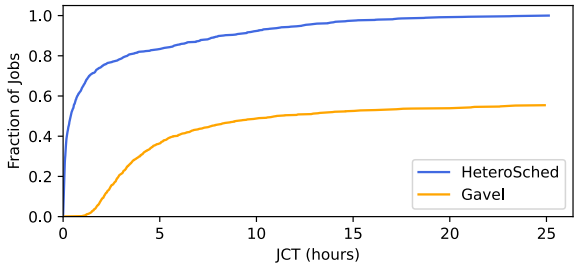
To isolate the benefit of plan-aware scheduling, `HeteroSched` is compared with fixed-family schedulers (DP+PP, TP+PP, and PP) and their corresponding decoupled variants. In the decoupled variants, the scheduler allocates GPUs using the throughput for that fixed parallelism family; Metis then selects the best plan executable on exactly that allocation without revising it. Thus, both approaches use Metis for parallelization, but only `HeteroSched` exposes plan quality during resource allocation. Figure 8 reports results on 24- and 48-GPU heterogeneous clusters.

The decoupled baseline recovers part of the fixed-family loss. PP is the strongest decoupled baseline in all 12 settings, and the decoupled baseline reduces its makespan by 17.7% on average. `HeteroSched` further reduces makespan by 23.4% on average (15.4–30.3%), so the strongest decoupled baseline requires 1.31× makespan of `HeteroSched`. The residual reduction averages 22.1% on 24 GPUs and 24.6% on 48 GPUs. The decoupled baseline can improve a chosen allocation, but it cannot recover from a poor allocation itself; exposing plan quality during scheduling therefore provides a distinct benefit.

The right panels of Figure 8 report GPU-time–weighted strategy shares by model size, so they capture where cluster capacity is

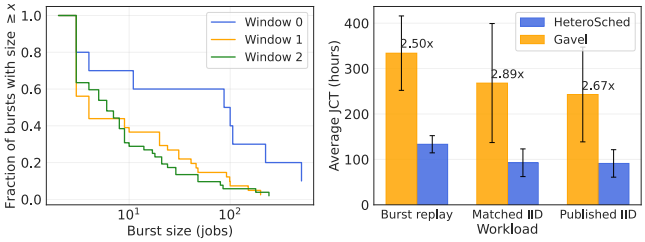


(a) Average job completion time (JCT) under varying job arrival rates.



(b) CDF of job completion times at 2 jobs/hour job arrival rate.

Figure 6: Average job completion time (JCT) comparison between `HeteroSched` and Gavel [22] on the 24-GPU heterogeneous Mixed workload under continuous job arrivals. Means are over three seeds; shading shows standard deviation.



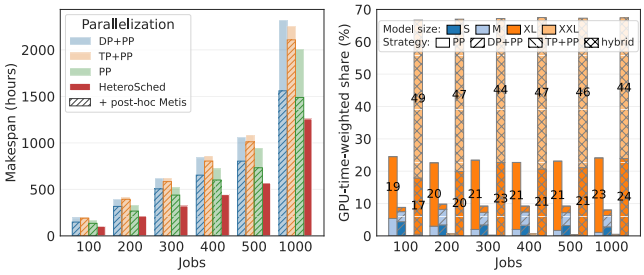
(a) Inferred burst-size CCDF. **(b) Average JCT.**

Figure 7: Robustness to workload structure. (a) Burst-size CCDFs for three replay windows containing 10, 41, and 52 bursts, with median sizes of 49, 3, and 5 jobs, respectively. (b) Mean JCT under structured replay, matched-IID, and published-IID workloads, averaged over three windows and three seeds. Error bars show standard deviation; labels report the ratio of mean Gavel JCT to mean `HeteroSched` JCT.

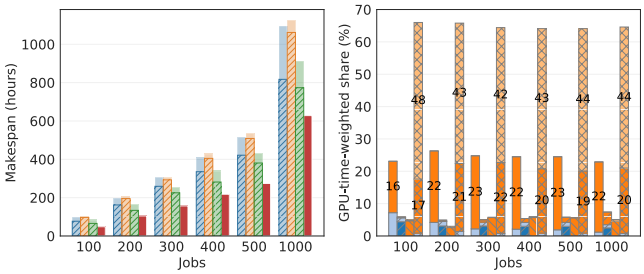
actually spent rather than simple job counts. They show a clear size-driven progression: small models are scheduled mainly with DP+PP, medium models rely mostly on PP, XL models are split between PP and hybrid plans, and XXL models are dominated by hybrid TP+DP+PP strategies. When the cluster grows from 24 to 48 GPUs, TP+PP becomes more visible for XL workloads, but hybrid remains the preferred choice for the largest models. Overall, this breakdown illustrates why exposing plan quality during allocation is useful: the preferred parallelization family changes substantially with model size and available resources. `HeteroSched` can account for these

Table 2: Effect of wait-time weighting under the average-JCT objective using the Figure 6 workload and cluster (Mixed workload excluding XXL jobs, 24-GPU heterogeneous cluster, three seeds). Aging uses w_i from Equation 6; No Aging sets $w_i = 1$. Tail/starvation metrics report mean ± std.; JCT is averaged over completed jobs. "Starved" denotes measured jobs unfinished at the predeclared 30-day horizon after the last measured job arrives. The $\lambda = 5$ setting is overloaded.

λ	Aging	p99 wait (h)	Max wait (h)	Starved	S JCT (h)	M JCT (h)	L JCT (h)	XL JCT (h)	XL completed (%)
2	Yes	7.57±1.07	12.28±1.93	0	0.15	0.34	1.90	10.63	100
2	No	8.18±2.02	20.00±4.05	0	0.09	0.15	0.93	10.93	100
5	Yes	647±72	774±24	3.0±1.7	0.67	2.28	13.75	194.24	95.3
5	No	807±4	826±7	29.7±6.7	0.09	0.19	1.50	41.68	53.9



(a) Cluster Size: 24 GPUs



(b) Cluster Size: 48 GPUs

Figure 8: Fixed-family, decoupled (post-hoc Metis), and HeteroSched scheduling on (a) 24-GPU and (b) 48-GPU heterogeneous clusters. Left: makespan for DP+PP, TP+PP, and PP baselines, their post-hoc Metis variants, and HeteroSched. Decoupled variants optimize the plan after GPU allocation is fixed. Right: HeteroSched's GPU-time-weighted strategy shares by model size; hybrid denotes TP+DP+PP.

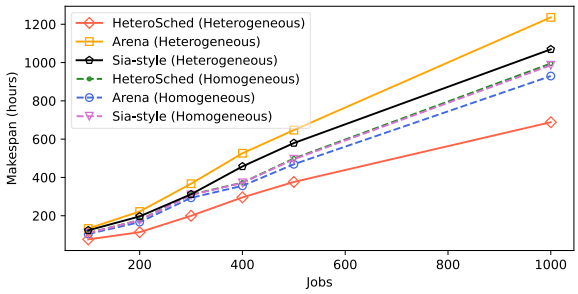


Figure 9: Comparison with integrated schedulers on the static Mixed workload. Arena and the Sia-style baseline search only homogeneous per-job plans, while HeteroSched also supports heterogeneous mixed-GPU allocations.

differences while choosing the resource allocation itself, whereas the decoupled baselines can only optimize the parallelization plan after that allocation has already been fixed.

4.4 Against Existing Integrated Schemes

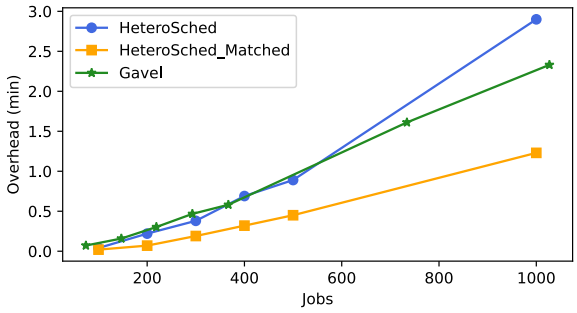
We next compare HeteroSched against two integrated baselines on the static Mixed workload: Arena [36] and a Sia-style scheduler [9]. The experiments use a 24-GPU cluster with three 8-GPU nodes, 5-minute scheduling rounds, and workloads ranging from 100 to 1,000 jobs. Both baselines can vary the number of GPUs assigned to a job and search over homogeneous per-job parallelization plans.

To isolate scheduler behavior, we match the workload, cluster configuration, and round length across all systems. Arena and the Sia-style baseline are additionally evaluated with the same homogeneous per-job plan space, while HeteroSched retains its heterogeneous mixed-GPU strategies. For the Sia-style baseline, we implement its published scheduling principles in our simulator and restrict its strategy search to homogeneous GPU groups, i.e., a single GPU type per job, consistent with Sia's design. For Arena, we use its simulator and evaluate it under the same workload and homogeneous-plan search space used for the Sia-style baseline. This avoids confounding effects from runtime differences and enables a controlled comparison under heterogeneous cluster conditions.

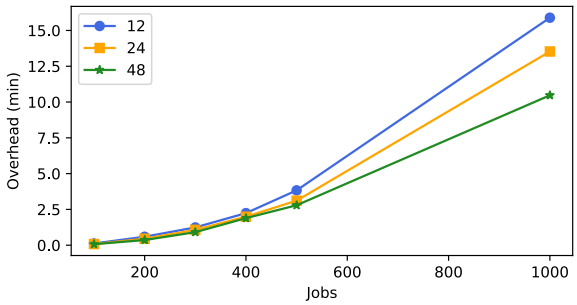
Figure 9 shows a clear split between the homogeneous and heterogeneous settings. On the homogeneous cluster, Arena is slightly better in makespan, outperforming the Sia-style baseline and HeteroSched by about 5% across workload sizes. This behavior is consistent with Arena's design, which is optimized for homogeneous per-job resource bundles. In this setting, HeteroSched and the Sia-style baseline remain very close, as expected given Sia's design. On the heterogeneous cluster, however, the ranking reverses. HeteroSched achieves the lowest makespan at every workload size, reducing makespan by 35–42% relative to the Sia-style baseline and by 38–45% relative to Arena. The key reason is that, although Arena co-designs scheduling and adaptive parallelism, its scheduler still allocates homogeneous bundles and therefore cannot exploit mixed GPU-type subsets within a job. As load increases, this restriction leads to missed placements, longer queuing, and lower cluster efficiency when available resources are fragmented across GPU types. In contrast, HeteroSched schedules over full job-strategy pairs, allowing it to combine heterogeneous GPU subsets with hybrid

DP/TP/PP plans, which yields substantially better cluster-level performance in the heterogeneous setting. Overall, these results show that integrating scheduling with adaptive parallelism is not sufficient by itself. Under heterogeneous clusters, the scheduler must also optimize over heterogeneous per-job resource subsets, rather than restricting each job to homogeneous allocations and searching only within that reduced space.

4.5 Scalability, Overhead, and Effectiveness



(a) Compare with Gavel, MIXED, 12 GPUs



(b) Different Cluster Size, XXL

Figure 10: Scheduler overhead versus the number of jobs across various scheduling approaches.

We next quantify scheduler cost and scalability, sensitivity to round and reconfiguration parameters, and the optimality loss introduced by HeteroSched's hierarchical decisions.

Figure 10a presents the scheduling runtime of HeteroSched compared to Gavel as the number of jobs increases from 100 to 1000. As observed, our scheduler demonstrates scalable performance comparable to Gavel. Even under heavy workloads with 1000 concurrent jobs, HeteroSched incurs a total scheduler overhead of 3 minutes accumulated over the entire execution, accounting only for LP solving and priority updates. This is approximately 20% higher than Gavel, while remaining negligible relative to overall job execution time. Moreover, when parallelization is disabled and job strategies are matched to those used by Gavel (HeteroSched_Matched), our algorithm incurs lower scheduling overhead than Gavel, highlighting the scalability of the chunked LP solver. Figure 10b reports scheduler overhead for the XXL workload on clusters with 12, 24, and 48 GPUs. We can see that HeteroSched keeps overhead low across scales. We observe that cumulative overhead tracks makespan rather than job count: smaller clusters complete the workload in more rounds, incurring more LP solves and thus slightly higher total overhead, whereas larger clusters finish in fewer rounds, amortizing solver cost. This behavior indicates the algorithm scales efficiently with the number of jobs and GPUs; the per-round scheduling cost remains well below the scheduling round length across all evaluated scales.

Scalability. We also evaluate the fidelity and scalability of chunked LP by comparing chunked and global LP solutions on selected XXL (1,000-job) cases from Figure 5, including an additional 960-GPU setting; the results are reported in Table 3. The reported overhead includes both the LP-based allocation step and the lightweight round-level scheduling mechanism; the value in parentheses denotes the corresponding average per-round scheduling cost in milliseconds. Overall, the chunked formulation closely matches the global solve, yielding comparable allocations and objectives while reducing scheduling overhead by up to 10×, with less than 3% makespan deviation. In the 960-GPU XXL case, chunked LP incurs only 7.09 minutes of total scheduling overhead, or 276.5 ms per round, which is still far below the 5-minute scheduling interval. These results indicate that chunking preserves solution quality while keeping both the allocator and the scheduling control path practical as cluster scale approaches 1,000 GPUs.

Table 3: Chunked vs. global LP at scale. Reported overhead and makespan are for chunked LP; values in parentheses show average per-round scheduling cost (ms).

Workload	GPUs	Overhead, min (ms)	Ratio	Makespan, hours	Makespan Dev. %
Het_XXL	12	16.57 (11.4)	1.30	7146.47	0.10
Het_XXL	24	13.76 (15)	1.79	4464.39	-0.05
Het_XXL	48	10.49 (24.6)	2.16	2113.03	2.15
Het_XXL	960	7.09 (276.5)	1.1	128.01	0.00
Hom_SM	12	1.05 (38.4)	9.87	136.70	-1.42

Ratio = Global_overhead / Chunked_overhead
Dev = 100 × Chunked_makespan/Global_makespan −100.
Het = heterogeneous; *Hom* = homogeneous.

Each previously unseen job is profiled for 20 iterations on up to 8 homogeneous GPUs, and Metis is invoked once per pruned (job, GPU-subset) pair. Profiles and generated plans are cached and reused for subsequent jobs with the same job properties; changes in resource availability require only reselection among cached plans, not reprofiling. In the largest profiling case—the 1,000-job Mixed workload on the 48-GPU heterogeneous cluster—profiling and planning take 90 and 7 minutes, respectively, only 0.35% and 0.03% of makespan. For an uncached job whose expected remaining execution time is too short to amortize profiling, the scheduler can instead execute its submitted fixed plan, i.e., the user-provided GPU count and DP/TP/PP configuration, as a single strategy tuple. Adaptive profiling and replanning are enabled only when their predicted remaining-time savings exceed the profiling and reconfiguration costs.

Round length and reconfiguration cost sensitivity. Figures 11 and 12 evaluate sensitivity to round length and reconfiguration

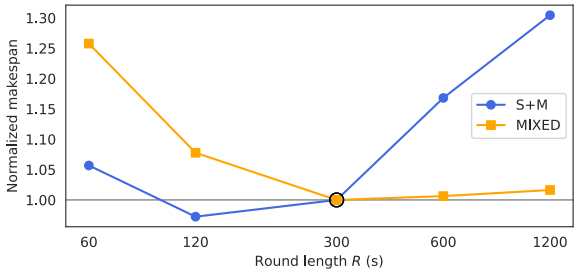
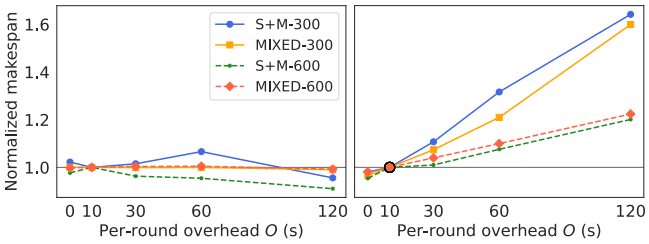


Figure 11: Round-length sensitivity for Small+Medium (S+M) and Mixed workloads on the 24-GPU heterogeneous cluster. Overhead is fixed at O=10 s; each curve is normalized to its R=300 s result.



(a) Overhead-removed makespan. **(b) Actual makespan.**

Figure 12: Reconfiguration-overhead sensitivity for Small+Medium (S+M) and Mixed workloads on the 24-GPU heterogeneous cluster at R=300 and 600 s. (a) Makespan excluding directly charged overhead, $M_{\text{net}} = M - N_r O$, where N_r counts executed rounds. (b) Actual makespan. Each curve is normalized to its corresponding O=10 s result.

overhead using 500-job static Small+Medium (S+M) and Mixed workloads on the 24-GPU heterogeneous cluster. For each workload, the same jobs and retained strategies are reused at every point. Figure 11 normalizes makespan to the default $R = 300$ s and $O = 10$ s. Both Small+Medium and Mixed workloads exhibit a U-shaped trend: Mixed is minimized at $R = 300$ s, while Small+Medium is minimized at $R = 120$ s, with the default only 2.8% higher. This behavior reflects two opposing effects. As R decreases, the number of rounds increases, magnifying both the fixed cost $O \times N_r$, where N_r is the number of executed rounds, and losses from per-round iteration flooring. As R increases, more capacity is stranded when jobs finish before the end of a round. Figure 12 stress-tests O at $R = 300$ s and 600 s, normalizing each curve to its $O = 10$ s setting. Figure 12b shows that, at $R = 300$ s, increasing O to 30 s raises makespan by 7.4% for Mixed and 10.7% for Small+Medium. At $O = 120$ s, these penalties reach 59.9% and 64.2%, respectively, but decrease to 22.4% and 20.1% at $R = 600$ s. Figure 12a reports the overhead-removed decomposition, $M_{\text{net}} = M - N_r \times O$, which varies by at most 1.3% for Mixed. This indicates that scheduling quality depends only weakly on reconfiguration overhead and that most of the observed degradation comes from accumulated per-round costs, which longer rounds effectively amortize.

Effectiveness. To quantify the loss from HeteroSched's hierarchical decisions, we construct a configuration-based oracle. Let C be the set of feasible configurations, where each configuration assigns at most one strategy to each job while satisfying all GPU-type capacities. Let $\text{thpt}_{i,c}$ denote job i's throughput under configuration c (zero if unscheduled), t_c the training time spent in c, and n_c the number of rounds using it. Using the same round length R, reconfiguration cost O, and iteration requirements as the simulator, the oracle minimizes makespan:

$$T^* = \min_{t_c, n_c} \sum_{c \in C} (t_c + O n_c) \quad \text{s.t.}$$
$$\sum_c \text{thpt}_{i,c} t_c \geq \text{iter}_i, \qquad 0 \leq t_c \leq (R - O) n_c, \quad n_c \in \mathbb{Z}_{\geq 0}. \tag{7}$$

For small instances, we enumerate all feasible configurations and solve the formulation exactly. At larger scale, we use certified column generation with lower and upper bounds within 0.5%. Exact enumeration on 4–8-job instances validates these certificates: the exact optimum lies within the certified interval for every completed comparison. Table 4 decomposes the gap into: retention, the loss from limiting each job to its top-20 strategies; allocation+placement, the loss from replacing the oracle schedule with HeteroSched's global LP and greedy round-based realization; and chunking, the additional loss from replacing the global LP with the chunked LP. Total compares HeteroSched directly with the all-strategy oracle.

Table 4: Optimality gap on the 24-GPU Mixed workload. Values are guaranteed average/max gaps (%) over five seeds.

Jobs	Retention	Allocation+placement	Chunking	Total
100	1.00 / 1.48	14.98 / 17.14	0.38 / 0.83	16.87 / 19.58
300	0.90 / 1.36	16.03 / 18.02	1.68 / 4.02	19.48 / 20.73

Reported values use conservative combinations of the certified lower and upper bounds and therefore represent guaranteed gaps rather than ratios of point estimates. Candidate retention contributes only 1.00% and 0.90% on average for 100 and 300 jobs, respectively. The dominant loss comes from realizing the fractional allocation as per-round placements, with average gaps of 14.98% and 16.03%, while LP chunking adds only 0.38% and 1.68%. Overall, HeteroSched has guaranteed average gaps of 16.87% and 19.48% for 100 and 300 jobs. Thus, strategy retention and LP chunking incur little additional loss, while keeping online scheduling tractable. The oracle is optimal over the profiled strategy tuples available to HeteroSched; it does not certify Metis's internal parallelization search.

4.6 Physical and Cost-Model Validation

We validate our simulator on two physical clusters using the same workloads: 47 dynamic jobs with Poisson arrivals at 25 jobs/hour and 24 static jobs submitted simultaneously. Both contain LLaMA-2 1B and 7B training jobs and use 5-minute scheduling rounds.

On the homogeneous 8×A6000 cluster, we use TorchTitan [19] to execute the selected parallel configurations. As shown in Table 5, simulated average JCT and makespan differ from physical

Table 5: Physical-versus-simulated validation using the same static and dynamic workloads on homogeneous and heterogeneous clusters. All errors are below 9%.

Cluster	Trace	Metric	Physical	Simulation
8×A6000	Dynamic	Avg. JCT	37.3 min	34.1 min
8×A6000	Static	Makespan	2.66 h	2.50 h
L40S+V100	Dynamic	Avg. JCT	23.4 min	25.3 min
L40S+V100	Static	Makespan	2.34 h	2.26 h

Table 6: Strategy prediction accuracy on the physical L40S/V100 cluster. Model/GBS gives model size/global batch size; Plan gives parallelism degree (pp or dp) and micro-batch size (mb); mixed rows span both GPU types. Mean absolute error is 4.6%, with a maximum of 9.3%.

Model/GBS	GPUs	Plan	Pred./Meas. (s/iter)	Error (%)
1B/256	1V100	pp=1, mb=32	73.67/71.36	3.2
1B/256	2V100	dp=2, mb=16	38.27/36.71	4.2
1B/256	2L40S	dp=2, mb=8	24.52/23.09	6.2
1B/512	1V100	pp=1, mb=64	145.6/142.6	2.1
1B/512	2V100	dp=2, mb=32	74.22/72.59	2.2
1B/512	2L40S	dp=2, mb=16	48.04/45.78	4.9
7B/128	4L40S	pp=4, mb=64	31.07/28.42	9.3
7B/128	4L40S+4V100	pp=8, mb=128	22.67/21.54	5.2
7B/256	4L40S	pp=4, mb=128	59.35/55.87	6.2
7B/256	4L40S+4V100	pp=8, mb=128	42.21/41.14	2.6

measurements by less than 9%. On the heterogeneous three-node 4×L40S+4×V100 cluster, we use HARP [18] as the runtime to execute the selected plans. Because the physical validation is limited to eight GPUs and 1B/7B models, the best configurations are often single-family DP or PP plans; hybrid DP/TP/PP strategies become more relevant at larger scales. Table 6 therefore focuses on cost-model accuracy, comparing predicted and measured iteration times for 10 LLaMA-2 1B and 7B strategies, including mixed-GPU plans. The mean absolute prediction error is 4.6%, with a maximum of 9.3%, and the predictions preserve the ordering of all evaluated strategies. For the same workloads used above, Table 5 shows simulation errors of 3.4% for static makespan and 8.1% for dynamic average JCT. These results validate both the mixed-GPU cost estimates used for allocation and the end-to-end simulator under heterogeneous execution.

5 Related Work

We have already extensively evaluated our work against Gavel [22], Sia [9], and Arena [36]. Compared with Gavel, `HeteroSched` does not assume rigid job configurations; compared with Sia and Arena, it adopts a more general strategy-tuple interface that allows the scheduler to optimize over mixed GPU-type subsets.

In other work, schedulers like Pollux [27] and ElasticFlow [6] assume homogeneous clusters. They can scale the GPU count or batch size—Pollux dynamically adjusts batch sizes and GPU assignments for each job but does not account for hardware heterogeneity, limiting its effectiveness in mixed-GPU environments. ElasticFlow supports deadline-aware admission control using concave scaling profiles, but still abstracts away GPU type, focusing only on resource scaling and job deadlines. In contrast, `HeteroSched` incorporates both resource heterogeneity and fine-grained parallelism into its LP-based scheduling formulation. The well-known paradigms of moldable and malleable scheduling techniques have been explored in DL workload context by Fujiwara et al. [4], who showed performance benefits when jobs can adjust GPU count before or during execution. However, their work uses measured data-parallel-style speedup curves, but does not model tensor- or pipeline-parallel tradeoffs. `HeteroSched` generalizes this model by integrating strategy search via Metis to support non-linear scaling across all three parallelism axes, guided by profiling and cost modeling.

There has also been a considerable amount of work on parallelizing DL workloads [11, 20]. Recent systems such as Piper [32], Alpa [38] and Metis [33] (which we use) automate the parallelization of individual deep learning jobs by searching over combinations of data, pipeline, and tensor parallelism strategies. Alpa introduces a hierarchical optimization framework that jointly optimizes intra- and inter-operator parallelism, while Metis extends automatic parallelization to heterogeneous GPU clusters by efficiently pruning the search space and adapting load balancing across diverse GPU types. However, both Alpa and Metis optimize parallelization under the assumption of dedicated resources for a single job and do not address scheduling across multiple competing jobs in a shared cluster environment.

6 Conclusion

This paper introduced `HeteroSched`, a novel framework that jointly optimizes deep learning (DL) job parallelization and cluster scheduling in heterogeneous GPU environments. `HeteroSched` unifies adaptive parallelism and heterogeneity-aware scheduling by dynamically generating a diverse set of parallelization strategies for each job and incorporating them into a parallelization-aware linear programming (LP) formulation. By explicitly modeling cluster and workload heterogeneity, `HeteroSched` identifies high-throughput configurations at runtime and schedules jobs as pairs (job, strategy). To ensure scalability, it employs a chunked/parallel LP solving procedure that efficiently explores the expanded scheduling space. Extensive evaluation on a range of workloads demonstrates that `HeteroSched` significantly improves key cluster performance metrics—including makespan and average job completion time (JCT)—achieving up to a 3× reduction in average JCT over Gavel and a 40% improvement in makespan relative to Sia. Moreover, comparison with a naive planner–scheduler decoupling shows that post-hoc parallelization cannot recover the loss from resource allocations made without plan awareness: even the strongest decoupled baseline requires 1.31× `HeteroSched`'s makespan on average. These improvements hold under both static and continuous job arrival scenarios, underscoring the effectiveness of tightly integrating adaptive parallelization with heterogeneity-aware scheduling.

Acknowledgments

This work was supported in part by the National Science Foundation under Grants CNS-2341378, CCF-2333895, and CCF-2428108.

References

[1] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/6c340f25839e6acdc73414517203f5f0-Paper.pdf

[2] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf

[3] Dror G. Feitelson and Larry Rudolph. 1996. Towards Convergence in Job Schedulers for Parallel Supercomputers. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS '96)*. Springer-Verlag, Berlin, Heidelberg, 1–26.

[4] Ikki Fujiwara, Masahiro Tanaka, Keniiro Taura, and Kentaro Torisawa. 2018. Effectiveness of Moldable and Malleable Scheduling in Deep Learning Tasks. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. 389–398. doi:10.1109/PADSW.2018.8644536

[5] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi11/dominant-resource-fairness-fair-allocation-multiple-resource-types

[6] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. ElasticFlow: An Elastic Serverless Training Platform for Distributed Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 266–280. doi:10.1145/3575693.3575721

[7] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: a GPU cluster manager for distributed deep learning *(NSDI'19)*. USENIX Association, USA, 16 pages.

[8] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: efficient training of giant neural networks using pipeline parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, Article 10, 10 pages.

[9] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 642–657. doi:10.1145/3600006.3613175

[10] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 947–960. https://www.usenix.org/conference/atc19/presentation/jeon

[11] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia (Eds.), Vol. 1. 1–13. https://proceedings.mlsys.org/paper_files/paper/2019/file/b422680f3db0986ddd7f8f126baaf0fa-Paper.pdf

[12] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3079856.3080246

[13] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[15] Adarsh Kumar, Kausik Subramanian, Shivaram Venkataraman, and Aditya Akella. 2021. Doing more by doing less: how structured partial backpropagation improves deep learning clusters. In *Proceedings of the 2nd ACM International Workshop on Distributed Machine Learning* (Virtual Event, Germany) *(DistributedML '21)*. Association for Computing Machinery, New York, NY, USA, 15–21. doi:10.1145/3488659.3493778

[16] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. AlloX: compute allocation in hybrid clusters *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 31, 16 pages. doi:10.1145/3342195.3387547

[17] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. 2021. TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 6543–6552. https://proceedings.mlr.press/v139/li21y.html

[18] Antian Liang, Zhigang Zhao, Kai Zhang, Xuri Shi, Chuantao Li, Chunxiao Wang, Zhenying He, Yinan Jing, and X Sean Wang. 2026. HARP: Orchestrating Automated Parallel Training on Heterogeneous GPU Clusters. In *Proceedings of the 21st European Conference on Computer Systems*. 1160–1180.

[19] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. 2024. TorchTitan: One-stop PyTorch native solution for production ready LLM pre-training. arXiv:2410.06511 [cs.CL] https://arxiv.org/abs/2410.06511

[20] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 2430–2439.

[21] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3341301.3359646

[22] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 481–498. https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak

[23] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 58, 15 pages. doi:10.1145/3458817.3476209

[24] NVIDIA. 2022. Meta Works with NVIDIA to Build Massive AI Research Supercomputer. Online. https://blogs.nvidia.com/blog/meta-ai-supercomputer-dgx/?utm_source=feedburner&utm_medium=email Accessed: March 2025.

[25] NVIDIA. 2024. NVIDIA Blackwell Architecture. Online. https://resources.nvidia.com/en-us-blackwell-architecture?ncid=pa-srch-goog-125782 Accessed: Feb 2025.

[26] Taylan Özden, Tim Beringer, Arya Mazaheri, Hamid Mohammadi Fard, and Felix Wolf. 2023. ElastiSim: A Batch-System Simulator for Malleable Workloads. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 40, 11 pages. doi:10.1145/3545008.3545046

[27] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 1–18. https://www.usenix.org/conference/osdi21/presentation/qiao

[28] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. doi:10.1145/3394486.3406703

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

[30] Nikko Ström. 2015. Scalable Distributed DNN Training Using Commodity GPU Cloud Computing. (2015). https://www.amazon.science/publications/scalable-distributed-dnn-training-using-commodity-gpu-cloud-computing

[31] Abeda Sultana, Fei Xu, Xu Yuan, Li Chen, and Nian-Feng Tzeng. 2024. Hadar: Heterogeneity-Aware Optimization-Based Online Scheduling for Deep Learning Cluster. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 681–691. doi:10.1109/IPDPS57955.2024.00066

[32] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. 2021. Piper: Multidimensional Planner for DNN Parallelization. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin,

P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 24829–24840. https://proceedings.neurips.cc/paper_files/paper/2021/file/d01eeca8b24321cd2fe89dd85b9beb51-Paper.pdf

[33] Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, Mohd Muzzammil, and Myeongjae Jeon. 2024. Metis: Fast Automatic Distributed Training on Heterogeneous GPUs. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 563–578. https://www.usenix.org/conference/atc24/presentation/um

[34] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[35] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. 2021. GSPMD: general and scalable parallelization for ML computation graphs. *arXiv preprint arXiv:2105.04663* (2021).

[36] Chunyu Xue, Weihao Cui, Quan Chen, Chen Chen, Han Zhao, Shulai Zhang, Linmei Wang, Yan Li, Limin Xiao, WeiFeng Zhang, Jing Yang, Bingsheng He, and Minyi Guo. 2026. Arena: Efficiently Training Large Models via Dynamic Scheduling and Adaptive Parallelism Co-Design. In *Proceedings of the 21st European Conference on Computer Systems* (McEwan Hall/The University of Edinburgh, Edinburgh, Scotland UK) *(EUROSYS '26)*. Association for Computing Machinery, New York, NY, USA, 89–107. doi:10.1145/3767295.3803571

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

[38] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin