关注在 dynamic resource availability + 异构资源下 model 角度出发的动态 parallelization plan 和 resource allocation

Sailor: Automating Distributed Training over Dynamic, Heterogeneous, and Geo-distributed Clusters

Foteini Strati* ETH Zurich Switzerland
Zhendong Zhang ETH Zurich Switzerland
George Manos ETH Zurich Switzerland
Ixeia Sánchez Périz† MIT USA
Qinghao Hu MIT USA
Tiancheng Chen ETH Zurich Switzerland
Berk Buzcu HES-SO Switzerland
Song Han MIT USA
Pamela Delgado HES-SO Switzerland
Ana Klimovic ETH Zurich Switzerland

Abstract

The high GPU demand of ML training makes it hard to allocate large homogeneous clusters of high-end GPUs in a single availability zone. Leveraging heterogeneous GPUs available within and across zones can improve throughput at a reasonable cost. However, training ML models on heterogeneous resources introduces significant challenges, such as stragglers and a large search space of possible job configurations. Current systems lack support for efficiently training models on heterogeneous resources. We present Sailor, a system that automates distributed training over heterogeneous, geo-distributed, and dynamically available resources. Sailor combines an efficient search space exploration algorithm, accurate runtime and memory footprint simulation, and a distributed training framework that supports different types of heterogeneity to optimize training throughput and cost.

1 Introduction

GPUs are in high demand for large-scale Machine Learning (ML). As ML models continue to grow exponentially in size, they require an increasing number of GPUs to train and fine-tune. This high demand makes it difficult for model developers to allocate the desired number of accelerators to train models at high throughput in public clouds or enterprise clusters [50, 56, 67]. Datacenters typically host a variety of GPU types and generations, spread across geographic regions [9, 21, 22, 60, 64]. Yet model developers tend to restrict model training to *homogeneous* clusters of GPUs, since state-of-the-art distributed training frameworks like Megatron-LM [48] and DeepSpeed [11] assume homogeneous GPUs and inter-node bandwidth. The demand for large, homogeneous GPU clusters compounds the scarcity of high-end GPUs.

Allowing a training job to run on heterogeneous GPU types and/or GPUs distributed across zones (i.e., with heterogeneous inter-node bandwidth) can give model developers access to *more GPUs* per job to increase training throughput. For example, consider a model developer who seeks to maximize training throughput by using 32 A100 GPUs (c2 in Figure 1), but discovers that only 16 A100s (c0 in Figure 1) are currently available in one zone. Using heterogeneous GPU generations (e.g., c3 uses an additional 16 V100s in the zone) or multi-zone configurations (e.g., c4 uses 32 A100s spread across 2 zones within a region) increases throughput by 1.15× and 1.87×, respectively, with a moderate increase in monetary cost per iteration (black line).

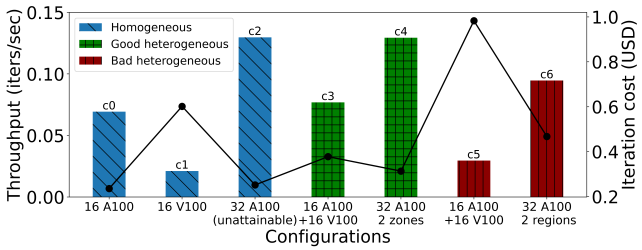


Figure 1. When homogeneous resources are limited, using heterogeneous GPUs (A100, V100) or multiple zones can increase OPT-350M training throughput (c3-c4) at low cost (black line). However, when the resource topology and job parallelization are not well selected, iteration time and monetary cost may increase significantly (c5-c6).

However, supporting heterogeneous, geo-distributed resources introduces several challenges. First, heterogeneous GPU types and placements across zones exponentially expand the configuration search space. Searching for an optimal configuration requires jointly optimizing the *resource allocation* and *job parallelization plan* (e.g., data/pipeline/tensor parallelism degrees). For example, in Figure 1, although c5 uses the same number of GPUs as c3, it achieves much lower throughput due to a suboptimal parallelization plan. The search must also consider the cost caused by extra resources and data transfers. Cloud providers charge significant fees for inter-zone and inter-region data transfers [4, 5, 10], which impacts geo-distributed configurations. In Figure 1, c6 uses the same number of GPUs and parallelization strategy as c4, but spreads training across regions (instead of zones within the same region), which increases cost.

Furthermore, as resource availability changes dynamically in datacenters, due to variable demand and node failures or preemptions, it is necessary to navigate this vast search space

*Correspondense to foteini.strati@inf.ethz.ch
†Work done while in ETH Zurich

quickly [54, 57]. Figure 2 shows the varying number of A100 GPUs that we were able to allocate over an 8-hour period, out of the 8 A100s that we continuously requested in 2 different zones in Google Cloud. Such changes in resource availability require frequently re-evaluating the job configuration search space. Thus, model developers require a system to quickly navigate the large search space of heterogeneous (and homogeneous) configurations with dynamic resource availability, optimizing for the user's performance-cost objective, while satisfying constraints (e.g., budget limits).

Second, profiling many candidate configurations to evaluate their throughput is prohibitively expensive and time-consuming. Hence, it is critical to accurately estimate the iteration time of a candidate configuration. This is challenging in heterogeneous environments, as differences in peak FLOPS, memory capacity, CPU-GPU interconnects, number of GPUs per node, and inter-node bandwidth typically lead to *stragglers*, which can significantly limit throughput. More importantly, variability in memory capacity per GPU may cause *out-of-memory (OOM) errors* in some GPUs, disrupting the entire training job. Simulating iteration time and checking that job configurations are valid (i.e., will not cause OOM) requires correctly modeling stragglers and per-GPU memory footprints.

Finally, after finding an appropriate resource allocation and parallelization plan for a training job given the available resources, model developers need to be able to run this job configuration in a distributed training framework (such as Megatron [48]). We find that optimal configurations for jobs running on heterogeneous resource topologies often include heterogeneous parallelism degrees per stage to load-balance the compute and memory capacity of different GPU types. Today's state-of-the-art distributed training frameworks need to be adapted to support such heterogeneous job configurations. Furthermore, as resource availability can change frequently (e.g., when using spot instances [54]), the training framework must be able to quickly reconfigure jobs.

Existing systems do not adequately solve these challenges. First, current works do not co-optimize the resource allocation with the job parallelization plan. Instead, systems like Aceso [28], Galvatron [32], and others in Table 1 expect the user to select a fixed resource allocation for which the system recommends a job parallelization plan. Most systems also do not consider heterogeneous resource topologies. Recent systems like Atlas [36], DTFM [68], Metis [56], and FlashFlex [66] optimize parallelization for heterogeneous GPUs or geo-distributed setups, but they suffer from prohibitively long search times (up to hours for configurations with 10s of GPUs) [56], or suboptimal cost functions [66, 68], making them unsuitable for environments with dynamic resource availability. Second, existing systems rely on inaccurate simulators to estimate the training throughput and memory footprint of candidate configurations. For example, Varuna [3] overlooks significant memory sources (e.g., memory needed by the optimizer, communication, etc) when estimating memory footprint, hence recommending configurations that lead to OOM errors. Finally, state-of-the-art distributed training frameworks like Megatron-LM [48] are slow to reconfigure jobs and do not support heterogeneous job parallelization plans or different microbatch sizes per GPU, which is necessary to maximize throughput and minimize cost in heterogeneous clusters.

To this end, we propose *Sailor*, a system for efficient large-scale training over heterogeneous resources with dynamic availability. Sailor consists of three components: a configuration planner, a simulator, and a distributed training framework. The Sailor planner navigates the search space of resource allocations and job parallelization plan combinations. It recommends configurations that optimize a user-defined objective (e.g., max throughput or min cost) under constraints (e.g., max budget or min throughput). The planner considers heterogeneous GPU and machine types and geo-distributed setups. The planner uses the simulator to accurately model iteration time and memory footprint for any given configuration. Through a combination of dynamic programming and search space pruning with effective heuristics, the planner finds solutions within seconds even for allocations with 100s of GPUs and varying degrees of heterogeneity. This allows Sailor to quickly adapt plans based on resource availability. Finally, the Sailor training framework adds support for heterogeneous configurations to execute the planner's configurations. It also improves fault-tolerance and elasticity by quickly adapting to changes in resource availability with efficient error detection and kill-free job reconfiguration. Together, these components enable Sailor to efficiently automate large-scale training in homogeneous, heterogeneous, and/or dynamic resource environments.

We evaluate Sailor in various setups and compare it extensively to prior works. To the best of our knowledge, our work is the first to compare the major open-source ML training planners proposed to-date (Table 1) in homogeneous and heterogeneous scenarios. We show that Sailor can find resource allocations and job parallelization plans that result in higher throughput than baselines in both homogeneous and heterogeneous clusters. We show how Sailor can leverage heterogeneous resources to improve throughput by 1.1-2.87× compared to the heterogeneity-aware baselines (Metis, FlashFlex, AMP), while maintaining search times of 10s of seconds compared to minutes or hours needed by the baselines. We also show Sailor's ability to increase performance and reduce cost in geo-distributed setups compared to DTFM by 5.9× and 9.8×, respectively. Finally, we demonstrate Sailor's ability to minimize monetary cost given throughput constraints, resulting in 40% cost savings compared to the second-best-performing baseline.

System	Support	Search Time (128 A100)
Piper [53]	3D, X, X, X	<1 sec
AMP [24]	3D, X, X, X	14 sec
Varuna [3]	3D, X, X, X	< 1 sec
Oobleck [20]	3D, X, X, X	hours
Metis [56]	3D, X, ✓, X	hours
FlashFlex [66]	3D, ✓, ✓, X	3 sec
Galvatron [32]	3D, X, X, X	10s of sec
Aceso [28]	3D, X, X, X	200 sec
DTFM [68]	2D, ✓, X, ✓	125 sec
Atlas [36]	3D, ✓, X, ✓	100 sec
Sailor	**3D, ✓,✓,✓**	**< 1 sec**

Table 1. Overview of distributed ML training planners. We omit planners that change ML training semantics [59]. The *Support* column stands for: [*degrees of parallelism supported, recommends resource allocation, supports heterogeneous GPU types, supports multi-zone*]. Search time assumes a cluster of 128 A100 GPUs and OPT-350M model.

2 Background

2.1 ML job parallelization strategies

Million or billion-parameter ML models train on massive datasets on clusters of high-end accelerators such as GPUs, using a combination of parallelization strategies:

Data Parallelism (DP): The model is replicated across workers, while the dataset is partitioned. At the end of each iteration, after each worker has computed the gradients on its own *minibatch*, the workers synchronize their gradients using an all-reduce collective [44].

Pipeline Parallelism (PP): The model is split into stages, with each stage consisting of a set of layers, and assigned to a worker or node. Workers operate at the granularity of a microbatch[1], performing forward and backward passes, sending activations to the next stage, and gradients to the previous stage. Due to inter-stage dependencies, pipeline parallelism is subject to *bubbles*, i.e., periods that a stage remains idle, waiting for others to complete [17]. As a result, many approaches have been proposed to process multiple microbatches simultaneously and reduce bubbles [3, 35].

Tensor Parallelism (TP): With tensor parallelism, a layer is divided across GPUs. After each GPU performs its local computations (both in forward and backward pass), the GPUs are synchronized using collectives such as all-reduce and all-gather. Since TP requires frequent communication, it requires very high interconnection bandwidths and is usually limited within a single node for reasonable throughput [48].

2.2 Automating parallelization strategies

Determining the optimal degree of parallelism for each dimension (DP, PP, TP) is complex and greatly affects training throughput. Multiple systems, which we refer to as *planners*, automate this process (see Table 1). Given a fixed resource allocation (e.g., 16 nodes with 4 A100 GPUs each), model configuration (including hyperparameters like global batch size and learning rate), model profiling information (e.g., time for forward and backward pass for different configurations), and hardware characteristics (e.g., network bandwidth), planners explore different parallelization strategies, estimating the training time under different configurations, using some form of simulation. Planners apply techniques such as exhaustive search [3, 56], dynamic programming [24, 53], and integer linear programming [71] to identify configurations that minimize training time.

[1]A minibatch is split in microbatches.

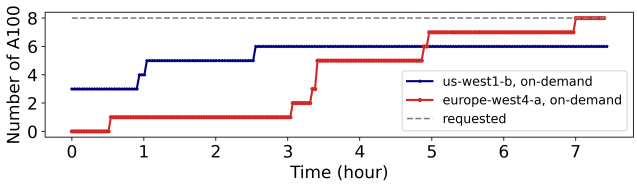


Figure 2. Availability of A100 GPUs in two zones in Google Cloud over 8-hr period. We request 8 GPUs in each zone. The trace was collected in April 2024.

3 Motivation and Challenges

3.1 Why use heterogeneous, geo-distributed GPUs?

Homogeneous high-end GPU clusters are scarce. The widespread adoption of ML, and hardware vendors' inability to keep up with this pace, has significantly increased GPU demand. Several studies report limited GPU availability across public cloud providers [7, 8, 50, 54, 67]. For example, Figure 2 plots the availability of on-demand A100 GPUs in two zones in GCP. In one zone, it took 7 hours to allocate 8 A100 GPUs, while in the other zone, the requested number of GPUs was not attained within the 8-hour window. Our findings align with the AWS GPU availability plot from Guo et al. [15], which shows that high-end GPUs such as A100 and H100 are difficult to aquire, while mid-tier GPUs (A10G, V100, T4) have higher but still limited and dynamic availability. Using GPUs across zones with heterogeneous types gives ML jobs the opportunity to use more GPUs to further increase throughput, as shown in Figure 1.

Power and cooling limits #GPUs per datacenter. From 2020 to 2025, the size of state-of-the-art ML models has increased by roughly 1200× [30, 61, 62], while per-GPU memory capacity has only increased by 8× [63]. This requires hyperscalers to deploy more and more GPUs [16, 55]. However, the high power and cooling requirements of high-end GPUs limits the amount that can be deployed per datacenter [49]. The next generation of ML models may need to train on GPUs across multiple availability zones [6, 27, 37, 46].

Using old GPUs can reduce embodied carbon. Embodied carbon (greenhouse gas emissions associated with manufacturing to disposal) is a major source of datacenter

emissions [1, 42, 58]. Although users prefer to allocate the latest GPUs for their ML jobs, older GPUs are abundant as the typical lifetime for ML servers in hyperscaler datacenters is ~6 years [45, 58]. Finding optimal ways to spread jobs across heterogeneous GPUs will enable leveraging older GPUs for longer to better amortize embodied carbon.

3.2 Challenges with Heterogeneous ML Clusters

C1: Quickly searching a vast configuration space. Considering heterogeneous and geo-distributed GPUs creates a vast and complex search space. The ML developer needs to decide how many GPUs to use and how to group them across VMs. This complexity increases further, when accounting for job parallelization plans within each allocation. Furthermore, the optimal allocation and partitioning strategies depend on user objectives and constraints. A planner needs to *quickly* navigate the large configuration space to adapt to dynamic resource availability in cloud and on-premise environments [14, 15, 54, 60], since maximizing throughput requires adapting parallelization strategies with changes to cluster topology [3, 57].

Table 1 shows that Metis [56], FlashFlex [66], Cephalo [15], Atlas [36], and DTFM [68] explore parts of this large search space. Atlas [36] and DTFM [68] target geo-distributed training, but do not consider heterogeneous GPU types, and do not decide the various parallelism degrees: instead, they take as input the parallelism degrees, and assign these degrees in the available zones. On the other hand, Metis [56], FlashFlex [66] and Cephalo [15], consider heterogeneous GPU types, but overlook geo-distributed training, and are quite inefficient for dynamic environments. Metis needs a few hours to devise a plan for a 16-GPU cluster (A100 and V100)[2], making frequent reevaluation infeasible as GPU availability changes. Cephalo [15] has a search time of 300 sec on a cluster of 64 GPUs[3], but it is limited only to Fully Sharded Data Parallelism. FlashFlex [66] has a short runtime, but provides inaccurate runtime estimations, leading to suboptimal plans. Furthermore, these planners only optimize for *throughput*, ignoring budget constraints, and cost (dollars per iteration), which affect the optimal configuration (§5.2.4).

C2: Accurately simulating memory footprint and iteration time. Most planners use analytical models or simulations to evaluate a configuration (parallelism degrees, microbatch sizes, etc) on a given cluster setup, since it is impractical and very expensive to deploy and profile every configuration. This evaluation usually includes two stages: 1) memory footprint estimation, to identify whether a configuration is valid (i.e., it does not lead to OOM errors) and 2) iteration time estimation, to determine performance.

[2]with max_permutation_length and device group variance set to 10 and 0.5 respectively, according to the paper [56]
[3]reported on the paper [15]

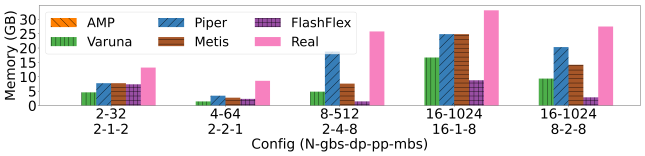


Figure 3. Peak Memory estimations of various baselines compared to the actual peak memory, for the OPT-350M model on a homogeneous cluster of 4 Grace-Hopper per node. N stands for the number of nodes, gbs is the global batch size, pp pipeline parallelism, tp tensor parallelism, dp data parallelism, and mbs the microbatch size.

Unfortunately, these estimations are often inaccurate, resulting in suboptimal or invalid plans. First, planners either completely ignore memory footprint [24], or underestimate the amount of memory requirements during training [3, 20], omitting activations, optimizer states, and memory fragmentation, or assume the training memory footprint is uniform across all devices and pipeline stages [26, 66]. As a result, these systems may find plans that cause OOM errors, when deployed. Figure 3 compares memory footprint estimations with the real footprint on a homogeneous cluster of up to 16 Grace-Hopper nodes for the OPT-350M model, showing that planners can be 25-95% off when estimating memory footprint. Second, planners often poorly model training time, due to incorrect assumptions about network bandwidth and ignoring communication-computation overlap [11, 69].

Modeling memory footprint and iteration time becomes even more complex with heterogeneity [33]. Different GPU generations vary in compute performance (e.g., TFLOPs), and memory capacity [72]. Thus, a configuration that fits in one GPU, might cause OOM errors in another GPU. Additionally, stragglers and network bandwidth differences (especially in geo-distributed setups [50]) must be considered for accurate timing estimations. Planners must accurately model compute, memory, and network bandwidth heterogeneity - yet, as shown in Table 1, most systems overlook this.

C3: Supporting both heterogeneous plans and seamless elasticity in a real distributed training framework. Most training frameworks, i.e., systems that train a model on a set of devices, assume homogeneous clusters and job configuration plans. For example, widely used and high-performing frameworks such as Megatron [48] and DeepSpeed [11] assume *uniform* parallelism degrees for the entire training job (e.g. DP=2, PP=6, TP=1). This limits efficiency in heterogeneous configurations, where various parallelism degrees per training subproblem (e.g., using PP=6, with first 3 stages having TP=4, and the next 3 stages having TP=2) help load-balance compute and memory on GPU nodes with different resources. Thus, a framework should accommodate heterogeneous degrees of parallelism. Furthermore, the framework should seamlessly reconfigure and adapt to dynamic GPU

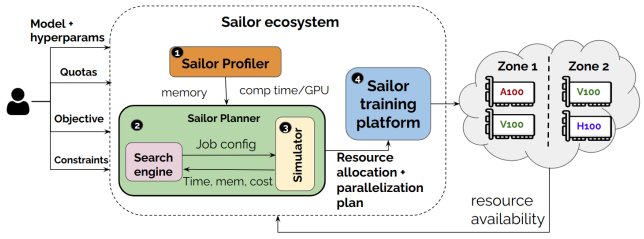


Figure 4. Sailor system overview.

availability (Figure 2), as long job reconfiguration times in response to resource changes are wasteful [54, 57]. Although related works propose elastic systems [3, 13, 54] or systems that support heterogeneity [31, 66], there is no open-source system that supports both.

4 Sailor

To address the above challenges, we propose *Sailor*, a distributed training ecosystem that consists of a profiler, planner, simulator, and distributed training framework. As shown in Figure 4, ML developers submit their model training specifications (model, optimizer, global batch size, etc), resource quotas (the maximum number of GPUs for each type and zone), an objective (e.g., maximize throughput or minimize cost), and optionally also constraints (e.g, a maximum budget per iteration or a minimum iteration time). Sailor also receives feedback about the current availability of hardware resources (which may be less than the quotas).

Workflow. The Sailor profiler (1) collects information about the training job, the compute nodes and network bandwidth (§4.1). The Sailor planner (2) uses this information to select a near-optimal *resource allocation* from the pool of available hardware and a *job parallelization plan* that optimizes the user's objective within the provided constraints (§4.2). The planner uses the simulator (3) to accurately evaluate various candidate plans (§4.3) in terms of throughput, memory footprint, and cost. Sailor then launches the job with the selected configuration using its distributed training framework (4) (§4.4), which is implemented on top of Megatron-DeepSpeed [11]. Sailor dynamically re-configures the job as resource availability changes.

4.1 Sailor Profiler

Training job profiling: When a user submits a training job, the Sailor profiler collects information about the job's compute and memory requirements. Sailor profiles a training job on a *single* GPU node for each different GPU node type in the available resource pool. To minimize profiling time and enable single-node profiling, the profiler reduces repeated layers to a single instance (e.g., it uses one transformer layer for a given LLM). We use PyTorch hooks [39] to collect information per layer: the time required for the forward pass, backward pass, and update phase with different microbatch sizes and tensor parallel degrees. We use CUDA Events for accurate GPU measurements [38]. We also track the number of parameters, output activation, and the memory required for intermediate stages per layer, using the PyTorch CUDA memory allocator [40]. The profiling overhead is negligible (a couple of minutes).

Cluster profiling: Sailor also collects information about the network bandwidth between any pair of different machine types. Since network bandwidth depends on the message size, Sailor collects network bandwidth measurements (using PyTorch collectives with NCCL backend) by varying the message size and fitting a polynomial function to get a set of coefficients for any pair of node types.

4.2 Sailor Planner

The Sailor planner takes as input the training job and cluster profiling information from the profiler, a performance or cost objective, and optionally a constraint such as a budget. The planner selects a resource allocation and a parallelization plan to optimize the objective under the constraints. The parallelization plan defines the number of pipeline stages P (which we refer to simply as *stages*), the data parallelism degree of each stage D, the D pairs of $(GPU_j, TP_j, Zone_j)$ for each stage, and the microbatch size mbs. The Sailor planner does not change the global batch size, thus it does not affect the job's training dynamics.

The combination of resource allocation and parallelization plan candidates creates a vast search space. To find efficient solutions *quickly*, Sailor: 1) prunes the search space with heuristics that consider training memory footprint, GPU capacity, and scalability constraints (§4.2.1), and 2) applies dynamic programming to reuse information about the performance of parallelization plans (§4.2.2). These techniques allow Sailor to find training plans in 10s of seconds even for large heterogeneous, geo-distributed scenarios (§5.2).

The planner iterates through different parallelism degrees and microbatch sizes (based on model characteristics and profiling information). For a given layer partitioning and microbatch size, it finds the tensor parallelism degree for each GPU type, based on memory constraints and scaling heuristics. Then, it iterates through all cloud region combinations and selects the data parallelism degrees to evaluate for a combination. For a fixed data parallelism degree, the planner invokes the $solve\_dp()$ function (Listing 1) that applies dynamic programming to determine the optimal stage configuration (§4.2.2). The planner considers the configuration valid only when it is within the user-specified constraint.

Finally, the planner sorts the configurations according to the objective and returns the best configuration.

4.2.1 Pruning the large search space with heuristics.

We introduce heuristics to prune the search, omitting cases early-on that would lead to suboptimal or invalid results:

H1: Limit tensor parallelism within a node. Tensor parallelism performance is known to degrade when spanning multiple nodes [48, 52]. We restrict tensor parallelism to a single node and do not explore cross-node pairs (unlike Metis [56]). As a result, each tensor parallel replica of a stage only uses a single GPU type.

H2: Prune OOM configurations early. Since each stage replica performs tensor parallelism only among GPUs of the same type, we can easily compute the *minimum* tensor parallelism degree of each GPU type, for a given pipeline parallel stage and microbatch size. We exclude cases with tensor parallelism below this minimum. The minimum tensor parallelism for each stage is independent of the number of available GPUs per type, so we can reuse it when resource availability changes.

H3: When maximizing throughput, consider data parallelism degrees in decreasing order, until throughput stops increasing. Sailor uses the same data parallelism for each stage. We observe that, with a fixed pipeline parallel degree, increasing data parallelism (by using more machines) benefits training throughput as more pipelines process minibatches independently. However, as the data parallelism degree increases, the time required for gradient synchronization also increases, negatively affecting training throughput. Thus, when optimizing for throughput, we first determine the maximum feasible data parallelism degree based on available resources, and then progressively reduce it, until throughput stops improving.

H4: When minimizing cost, consider data parallelism degrees in increasing order, until cost per iteration stops decreasing. Following the logic from *H3*, for a fixed pipeline, doubling the data parallelism degree will lead to doubling the number of resources, but will not half the iteration time (due to all-reduce scaling overheads). Thus, configurations with a lower data parallelism degree lead to lower cost/iteration. Thus, when the user objective is minimizing cost/iteration, we search for *increasing* data parallel degrees D until a solution within the throughput constraint is found.

H5: Keep data parallel communication within a single region, while spreading pipeline parallel communication across more than one region. As shown by earlier work [36, 50], data parallelism performs poorly across regions due to the low network bandwidth. We constraint all data parallel pairs of a stage within a single region.

H6: Treat multiple zones within the same region as a single zone. Within a cloud region, the network bandwidth across zones is similar to the network bandwidth within a zone [50]. Thus, to reduce the search space, we consolidate all zones in a region into a single zone and do the geo-distributed partitioning at a region granularity.

4.2.2 Selecting per-stage configurations with dynamic programming.

We now describe how we find optimal resource configuration per stage for a given pipeline and data parallelism degree, microbatch size, tensor parallel degrees per stage, and GPU type. The formulation we describe below assumes an iteration time minimization objective and §4.2.3 describes how we further incorporate cost constraints. For brevity, we omit the reverse optimization (monetary cost minimization under throughput constraints).

Problem formulation and goal: Given a pipeline parallel degree P, a microbatch size mbs, a data parallel degree D, and tensor parallel degrees tp_{ij} for GPU j and stage i, we want to find, for each stage i, the D replicas, where each replica is a tuple $(j, tp_{ij}, zone_k)$ that minimizes iteration time. Note that Sailor precomputes tp_{ij} (Heuristic *H2*).

Why dynamic programming? Optimizing resource allocation per stage is challenging, as assigning resources to one stage impacts overall runtime and availability for other stages. In a heterogeneous, multi-zone setup, the number of possible resource combinations per stage explodes. Related works on homogeneous clusters used Integer Linear Programming (ILP) [20, 71], exhaustive search [3], or dynamic programming [24, 53]. We adopt dynamic programming for its ability to decompose the problem into subproblems and reuse intermediate results.

Dynamic programming formulation: Assume we have a pipeline with degree P, and we want to solve the dynamic programming problem for stage i that has L layers, with $l0$ being the first layer. We formulate selecting a resource configuration per stage as finding r resources to give to stage i while minimizing the iteration time T_{iter} as follows:

$$T_{iter}[l_0][P][R] = min_r\{T_{total}(T_{iter}[l_0+L][P-1][R-r], T_i(r))\} \quad (1)$$

for a given pipeline parallel degree P, stage i, and available resources R. We give r resources to stage i, and $R - r$ resources are available for the subsequent stages. In Sailor, r represents a map of different GPU types and zones, and should be enough to get D replicas of this stage. Based on heuristic *H5*, we keep each stage within a single region. T_{total} is calculated by identifying the straggler between stage i and the rest pipeline, the pipeline communication cost between stage i and $i+1$, and the synchronization bottleneck between stage i and the rest pipeline.

Implementation: Listing 1 implements the above formulation for heterogeneous GPUs and various cloud zones and regions. The procedure starts by generating all resource combinations of different GPU types that could fit this stage with the specified data parallelism (line 2). Then, for each combination r, it finds the next available region that fits stage i with r resources (lines 6 and 14). If i is the last stage, it returns the configuration that minimizes the stage time.

4.2.3 Adding a monetary cost constraint.

So far, we have formulated the iteration time minimization as a dynamic programming problem, and we split the problem at per-stage subproblems as shown in Eq 1 and Listing 1. When introducing a budget constraint, we need to account for the

remaining budget per stage, to solve the DP subproblem for that stage. The monetary cost depends both on the resources used and the iteration time. However, the iteration time depends on the pipeline straggler, which is not yet known when solving the resource allocation subproblem for stage i. To overcome this limitation, when solving the subproblem for stage i, we approximate the remaining budget by first considering that stage i is the straggler, and solving for the remaining stages with the respective remaining budget. At the end, we determine the actual straggler. If our straggler assumption was not correct, we adjust the budget with the new straggler and we solve the subproblems again.

Problem formulation: The monetary cost per iteration includes the cost due to compute resources C_{comp} and due to data transfer C_{comm}. When introducing a monetary cost constraint C, the cost per iteration should satisfy:

$$C_{iter} = C_{comp} + C_{comm} <= C => \sum_{i=0}^{i=P-1} Ccomp_i \cdot T_{iter} + C_{comm} <= C$$

$$\sum_{0}^{P-1} Ccomp_i \cdot (\sum_{0}^{P-1} t_i + N_{batches} \cdot t_{straggler} + max_0^P(t_{sync_i})) + C_{comm} <= C$$

where $\sum_0^{P-1} t_i$ stands for the pipeline warmup and cooldown phase, $N_{batches}$ is the number of microbatches processed per pipeline, and t_{sync_i} is the time required for the synchronization of all replicas of stage i. Since large models usually train with large global batch sizes, the $N_{batches} \cdot t_{straggler}$ term usually determines the iteration time, we can rewrite as:

$$\sum_{0}^{P-1} C_{comp_i} \cdot (N_{batches} \cdot t_{straggler}) + C_{comm} <= C \qquad (2)$$

From a dynamic programming perspective, assuming we are at pipeline stage i, which has a maximum budget limit C_{cur}, the cost constraint will be: $C_i + C_{rem} <= C_{cur}$, where C_{rem} is the cost of the remaining stages, and C_i is the cost of stage i. From Eq. 2, we have that $C_i = C_{comp_i} \cdot (N_{batches} \cdot t_{straggler}) + C_{comm_i}$. In Listing 1, line 14, when exploring a resource combination r, we can easily find C_{comp_i}, and C_{comm_i} for stage i. Since we do not know $t_{straggler}$, which is required to specify the remaining budget for the next stages, we use the approximation in lines 17-32: We begin assuming stage i is the straggler ($t_{straggler} == t_i$), and compute the C_{rem} for the next stages accordingly. We call the *solve_dp* function for the next stages giving C_{rem} as the budget constraint (line 20). If a solution cannot be found, we proceed with the next resource combination for stage i (line 22). If a solution is found, we check whether it is within the cost limit and keep the one with the maximum throughput (line 26). We then check the straggler of the found solution: if it is the same with the one we assumed, we break, and proceed to the next resource combination (lines 27-28). Otherwise, we adjust the budget with the new straggler (lines 31-32) and iterate again.

Listing 1. Resource-Stage assignment algorithm

```
1  def solve_dp(P, i, R, D, tp_gpu_stages, Ci):
2      Rcombos = generate_combos(R,D,tp_gpu_stages[i])
3      if i==P−1:
4          time_i = None
5          for r in Rcombos:
6              next_region = find_region_fits(i, r, curr_region)
7              time_ir = time_for_stage(i,r,next_region)
8              cost_ir = cost_for_stage(i, r, next_region, time_ir)
9              if cost_ir <= Ci: time_i = min(time_i, time_ir)
10         T_iter[P][i][R] = time_i
11     else:
12         time_all = None
13         for r in Rcombos:
14             next_region = find_region_fits(i, r, curr_region)
15             time_i = time_for_stage(i,r,next_region)
16             cost_i = cost_for_stage(i, r, next_region, time_i)
17             C_rem = Ci − cost_i
18             assumed_straggler = time_i
19             while (C_rem > 0):
20                 nextconf = solve_dp(P, i+1, R−r,
21                     next_region, all_regions, C_rem)
22                 if nextconf is None: break
23                 time_all = total_time(time_i, nextconf.time)
24                 cost_all = total_cost(cost_i, nextconf.cost)
25                 if (cost_all <= Ci):
26                     time_all = min(time_all, time_i)
27                 if nextconf.straggler < assumed_straggler:
28                     break
29                 cost_i = cost_for_stage(i, r, next_region,
30                     straggler=nextconf.straggler)
31                 C_rem = Ci − cost_i
32                 assumed_straggler = nextconf.straggler
33         time_all = min(time_all, time_i)
34     return T_iter[P][i][R]
```

4.3 Sailor Simulator

The planner uses the simulator to evaluate the performance and memory footprint of the generated plans. The Sailor simulator takes as input a training job specification (model, global batch size, optimizer, hyperparameters) and a job parallelization plan. It then estimates the memory footprint per GPU, the iteration time, and the cost per iteration. The simulator allows the planner to easily specify different types of heterogeneity: hardware heterogeneity (GPU type, number of GPUs per node, and network bandwidth) and job configuration heterogeneity (number of pipelines and different stage configurations per pipeline). The training configuration also specifies the microbatch size. The simulator also incorporates the information collected by the profiler about the training job and network bandwidth of the used links.

Memory footprint estimation: The Sailor simulator accurately estimates a training job's memory footprint by: 1) calculating memory footprint per GPU, per-stage and 2)

considering *all* main sources of memory footprint during training. Compared to prior works that assume a homogeneous memory footprint per stage, we observe that for a given parallelism configuration, the memory footprint of a training worker depends on layer partitioning, its pipeline stage index, its tensor parallelism degree, and its microbatch size. Hence, memory footprint varies among different workers and needs to be analyzed per worker to detect OOM scenarios.

Second, the peak memory footprint of a worker throughout training consists of various sources, often ignored in prior works. The peak memory footprint M_{peak} of a worker is given by: $M_{peak} = M_{model} + M_{activation}$, where M_{model} corresponds to the memory needed to keep copies of model parameters and is given by $M_{model} = num\_params \cdot mul\_factor \cdot data\_type\_size$. num_params is found by the stage id and tensor parallelism degree of the worker, while mul_factor accounts for multiple copies needed for the model itself, the optimizer, gradients, and communication [41]. $M_{activation}$ is the memory needed for storing layer activations and depends on the stage id and tensor parallelism degree of a worker, and microbatch size. By computing the memory footprint for each worker and comparing with the worker's GPU capacity, the simulator can easily detect OOM cases.

Iteration time estimation: We define one iteration as a full pass over the user-defined global batch size. The iteration time is calculated as: $T_{iter} = \max(T_{ppi}) + T_{sync} + T_{update}$, where T_{ppi} is the time needed for pipeline i, T_{sync} is the time needed for gradient synchronization at the end of an iteration, and T_{update} is the time for model update. We compute T_{pp_i} and T_{sync} following the formulas described in [50] for 1F1B pipeline parallelism, using our profiling for network bandwidth with respect to the message size per network link, to estimate communication time (for peer-to-peer and collectives). For each pipeline, the 1F1B pipeline parallelism schedule includes a warm-up, steady, and cool-down phase per iteration, where the steady phase is determined by the stage with the largest computation time (straggler). After computing the iteration time per pipeline, we compute synchronization time. Taking the maximum time per pipeline accounts for straggler effects caused by heterogeneity in GPU generations, inter-GPU, and CPU-GPU interconnects.

Iteration cost estimation: Related works (Table 1) do not compute the monetary cost of different resource allocation and job configuration combinations, since they only optimize for throughput. However, a very important metric to account for is *cost per iteration*, especially with geo-distributed training, due to costs associated with across-zone communication. Since Sailor does not change the global batch size and training hyperparameters, the number of iterations needed to reach convergence is constant, regardless of the cluster setup and parallelization strategy. Thus, the monetary cost per iteration indicates the total budget needed for the whole training. The metric depends both on the cost of allocated resources, as well as the training throughput and communication cost. The cost per iteration is given by $C_{iter} = C_{comp} + C_{comm}$, where C_{comp} is the cost due to compute resources, and is calculated as $\sum_i (N_i \cdot cost\_per\_gpu_i) \cdot T_{iter}$, for all different GPU types i in the cluster. C_{comm} stands for the cost for data exchange per iteration (e.g., when using geo-distributed training in public cloud) and is given by $C_{comm} = \sum_{ij} (bytes_{ij} \cdot cost\_per\_byte\_ij)$ for all zones i, j in the cluster. $bytes_{ij}$ might include traffic for data and pipeline parallel communication.

4.4 Sailor Distributed Training Framework

The Sailor training framework receives the job configuration that the planner generates, sets up the cluster, and starts training. We modified Megatron-DeepSpeed [11] to support heterogeneous plans and seamless elasticity, to make it compatible to planner's output and allow for fast reconfiguration when resource availability changes.

Support for heterogeneous plans: We added support for varying tensor-parallel degrees across data-parallel pairs per pipeline stage. The framework takes as input a rank topology for each stage, allowing each rank to belong to distinct tensor-parallel groups. Different tensor parallelism per stage affects the pipeline and data parallel communication, requiring workers to split or replicate activations and gradients across multiple peers. To accommodate this, we adjust the PyTorch communication groups and modify the send/recv and all-reduce operations accordingly.

Support for seamless fault-tolerance and elasticity: The Megatron-DeepSpeed framework lacks failure recovery and dynamic resource reconfiguration: the whole training needs to stop, and the user needs to *manually* reconfigure and restart the job. However, resource availability frequently changes both in the cloud (especially with spot instances) [50, 54] and in on-premise datacenters as jobs start or finish [57]. We introduce modifications for fast, automated failure detection and reconfiguration, minimizing rollback time. Each job consists of a controller and multiple workers. The workers handle training, while the controller monitors their status and detects resource availability changes. Upon detecting a change, the controller reinvokes the planner to generate a new plan and instructs workers to adjust accordingly. We follow a kill-free approach to minimize the reconfiguration time, where the existing workers do not exit, but instead destroy the current communication group, clean up their GPU memory, repartition the model, and setup a new communication group. If additional resources become available, the controller waits for new workers to initialize before updating the training configuration. Training restarts from the latest available checkpoint. We use asynchronous checkpointing to minimize the rollback time [34, 51].

5 Evaluation

We evaluate Sailor to answer the following questions:

1. How accurately does the Sailor simulator estimate iteration time and memory footprint?
2. How well does the Sailor planner perform compared to baselines in homogeneous and heterogeneous setups, in terms of throughput and monetary cost?
3. How is the Sailor planner search time affected by the cluster size, resource heterogeneity, user constraints and the different optimizations?

System configurations: We evaluate Sailor using real hardware and simulations. We use different cluster setups and GPU generations, in both cloud environments and on-premise clusters. In the public cloud, we used VMs with A100-40GB and V100-16GB from GCP [9]. For our on-premise datacenter experiments, we used a cluster with up to 32 homogeneous machines with 4 Grace-Hopper GPUs each, and a cluster of heterogeneous machines with 2x8 Titan-RTX, 3x8 RTX-2080, and 2x8 RTX-3090.

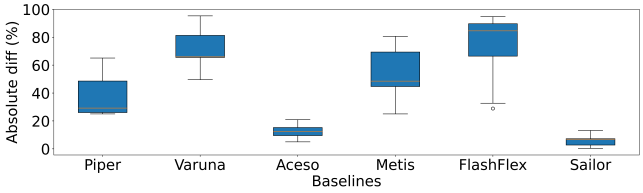
Models: We use the OPT-350M [19] and GPT-Neo-2.7B [18] models, with global batch size of 2048 sequences, and sequence length of 2048 tokens, with the Adam optimizer.

Baselines: To our knowledge, we present the first comprehensive comparison of major planners for large-scale ML training, including planners targeting *homogeneous* resources (Piper [53], AMP [24], Varuna [3], Aceso [28]), *heterogeneous* resources (Metis [56], FlashFlex [66]), and *geo-distributed training* (DTFM[68]). All baselines, except Aceso, are integrated into our platform with a unified Python API. We profile our models once and give each baseline its required profiling information. Aceso defines its own operators, which we profile separately. As Aceso also uses the Megatron backend, per-layer runtime profiles are very close to our models. We used the open-source version of all baselines. DTFM [68] does not determine parallelization strategies (e.g., DP, PP), but instead partitions a given plan. Therefore, we exhaustively generated all homogeneous parallelization plans and applied their partitioning methods to each. As the Atlas paper does not have an open-source implementation of runtime and memory simulation, we were unable to test Atlas end-to-end. However, we tested the zone assignment described in the paper [36], which performs similar to Sailor.

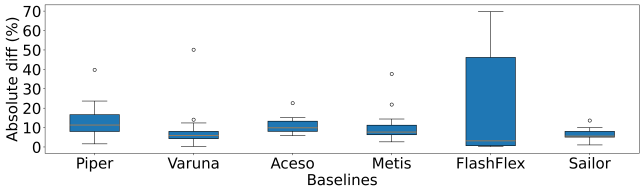
5.1 Validation of the Sailor simulator

We evaluate the accuracy of the Sailor simulator's iteration time and memory footprint estimations. We vary the number of devices and parallelization plans, find the difference in iteration time and peak memory footprint, and summarize using box plots. We omit AMP and DTFM since they do not support memory estimation.

Cluster of homogeneous GPU types. Figures 5a and 5b show the iteration time and peak memory footprint estimation error for the homogeneous cluster of Grace-Hopper for the OPT-350M model, respectively. Most baselines fail to accurately capture the peak memory footprint, since they ignore significant memory sources and assume a homogeneous memory footprint across the different pipeline stages. The examined baselines exhibit an error of 12.5-74% on average, while Sailor achieves an average error of 5.56%. Sailor also reduces the average runtime estimation error in the homogeneous setup to 6%, compared to 10-20% for the baselines.



(a) Peak memory footprint estimation



(b) Iteration time estimation

Figure 5. Different planners' compute and memory estimation on a cluster of GH200 GPU for the OPT-350M model.

Cluster of Heterogeneous GPU types. Figure 6 shows the iteration time prediction error for the OPT-350M model in a heterogeneous cluster of Titan-RTX, RTX-2080, and RTX-3090. The homogeneous planners (Piper, Varuna, Aceso) do not consider the differences in forward and backward passes of the different GPU types, resulting in an average error of 28%, 47%, and 37%, respectively. Even heterogeneous planners fail to accurately capture runtime: since FlashFlex relies on the theoretical performance of GPUs, it cannot accurately predict the runtime, getting an error of 69%. Metis fails to fully capture the heterogeneous network bandwidth between nodes, thus miscalculating the communication cost, resulting in 28% error in iteration time estimation, on average. In contrast, Sailor's iteration time estimations error is 4.5% on average.

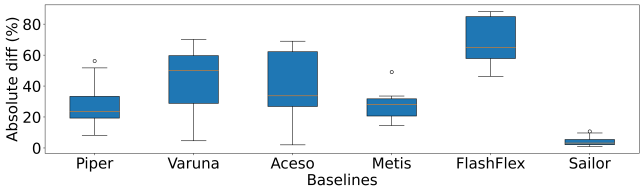


Figure 6. Different planners' iteration time on a cluster with heterogeneous GPUs for the OPT-350M model.

5.2 Sailor Planner vs. Baselines

We use our simulator to evaluate the throughput achieved by Sailor and the baselines across various cluster configurations. All baselines require a predefined resource topology as input: we consider 4-GPU VMs for each GPU type. Sailor takes resource quotas as input (total *number* of GPUs per type per zone) and jointly determines both the topology (VM allocation) and the parallelization plan. For Metis, we impose a 300-second time limit and use the best solution found within that period, if available. We summarize key takeaways.

5.2.1 Homogeneous Setups. Figure 7 shows the throughput achieved using the baseline planners and Sailor with only A100 GPUs for the OPT-350M model. Varuna failed to generate a valid plan that would not lead to OOM errors due to the poor memory estimation, and its limited search space (only supporting 2D parallelism). Sailor improves throughput by 1.15× compared to the closest baseline (DTFM), and even up to 5.7× (compared to Aceso).

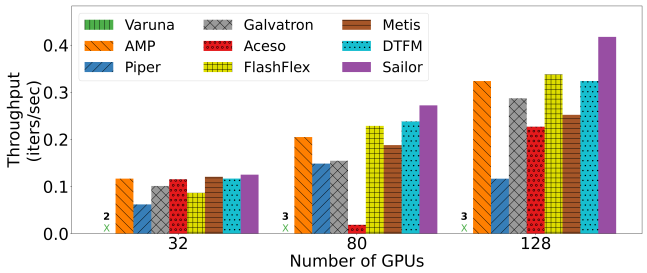


Figure 7. Comparison of the different planners considering A100-40GB GPUs for the OPT-350M model in one zone

5.2.2 Heterogeneous Setups. We evaluate Sailor's throughput compared to baselines (AMP, Metis, and FlashFlex) in a heterogeneous cluster setup using a mixture of A100 and V100 for OPT-350M (Figure 8) and GPT-Neo-2.7B (Figure 9). We also compare the throughput achieved by Sailor, when using only homogeneous resources (either A100 or V100). We vary the ratio and amount of A100 and V100 (50%-50% and 25%-75%) to assess the impact of having more low-end GPUs in the cluster [15]). As in the homogeneous setup, all baselines get a fixed resource topology (4-GPU VMs) as input, while Sailor also determines the resource topology along with the parallelization plan. We also report the monetary cost per iteration at each scenario (number on top of bar), and the number of plans generated by the baseline that would lead to OOM before a valid plan was found (**bold** number on top of bar).

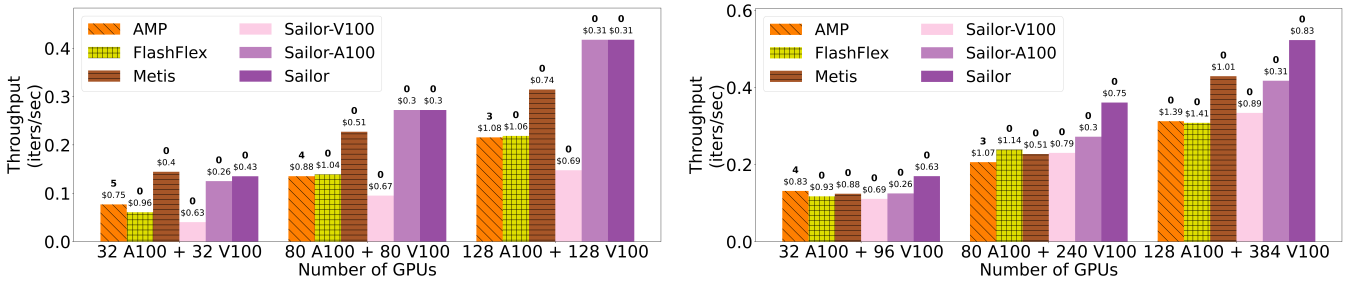
Throughput of heterogeneous baselines: Although AMP achieves high throughput in the homogeneous case, it performs poorly in the heterogeneous scenario, as it only allows for homogeneous plans, and does not correctly model stragglers. As a result, the monetary cost per iteration also increases. Furthermore, since AMP does not model training memory footprint, it leads to a large number of OOM plans, especially in the case of the GPT-Neo model (Figure 9).

FlashFlex achieves similar or higher throughput than AMP, as it can consider heterogeneous plans, and captures differences in compute between the different GPUs. However, its throughput is still low, as it uses low tensor parallelism and microbatch sizes. This leads to higher iteration costs (e.g. Figure 8), since it uses a large number of resources with a low throughput. It also fails to find valid plans for the large GPT-Neo model due to suboptimal memory estimation. Metis capped at 300 seconds achieves higher throughput than FlashFlex and AMP, due to more accurate runtime and memory footprint estimation, layer partitioning, load balancing, and exhaustive search of different GPU group combinations, but it generates a huge number of OOM plans (Figure 8).

Sailor achieves significantly higher throughput compared to baselines: for the OPT-350M model, Sailor achieves 1.9×, 2.03×, and 1.15× higher throughput compared to AMP, FlashFlex, and Metis, respectively, when the ratio of A100 and V100 GPUs is equal, and 1.57×, 1.55×, 1.39× higher throughput when more V100 are available. The speedups are similar for the GPT-Neo model as well. Also, Sailor does not output invalid plans, significantly improving the plan deployment compared to baselines with 10s of invalid plans. Sailor's ability to discover efficient resource topologies and parallelization plans also translates to significantly lower cost compared to the baselines (up to 2.67× lower cost compared to baselines in Figures 8 and 9).

Search times: Table 2 shows the search time for Figure 9b for the different baselines. Metis is capped by 300 sec in all cases, after when its search is interrupted. The other baselines are significantly faster, as they manage to finish the search in less than 200 sec for all cases. Sailor keeps the search time within 1 minute even in the largest cluster case of 512 GPUs, which demonstrates its efficient searching algorithm.

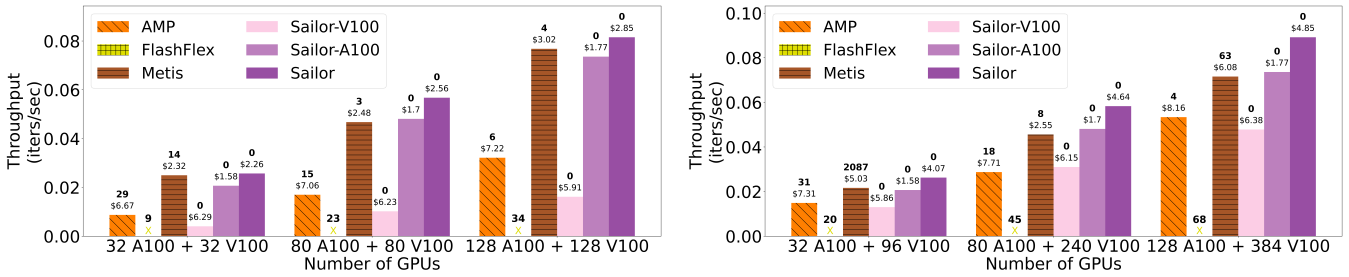
Benefits of heterogeneity: Figures 8 and 9 also show the throughput achieved by Sailor when given homogeneous-only setups of A100 (Sailor-A100) or V100 GPUs (Sailor-V100). Given that V100s are less efficient than A100s, using A100 only, or a mixture of A100 and V100 is always better than using V100 only. However, using V100 *in addition* to A100 does not always improve throughput. Heterogeneity is more beneficial when resources are limited (e.g. Figure 8a, with 32 GPUs per type), or with larger models like GPT-Neo. In fact, for the OPT-350M model, when 128 A100 and 128 V100 are available, Sailor chooses a plan with 128 A100 only, as it determines that no additional benefits will be gained by adding extra resources. Moreover, when the ratio of V100 to A100 is higher than 1, the throughput improvents are more significant, as shown in Figures 8b and 9b, where the V100:A100 ratio is 3:1. This aligns with the GPU memory capacity ratio, as well as the time for forward/backward pass for the transformer layers of the two GPU types, enabling



(a) 50% A100, 50% V100

(b) 25% A100, 75% V100

Figure 8. Comparison of planners considering A100 and V100 GPUs for the OPT-350M model in 1 zone



(a) 50% A100, 50% V100

(b) 25% A100, 75% V100

Figure 9. Comparison of planners considering A100 and V100 GPUs for the GPT-Neo-2.7B model in 1 zone

Baseline	A100-V100		
	32-96	80-240	128-384
AMP	31.22	51.43	86.41
FlashFlex	4.05	54.67	222.64
Metis	300	300	300
Sailor	1.6	7.67	17.4

Table 2. Search times (in sec) for Figure 9b.

better load balancing. Finally, heterogeneity leads to a higher cost, as more resources are used. Since the objective is to maximize throughput, Sailor ignored monetary cost when searching job configurations. In 5.2.4, we will show Sailor's ability to consider budget optimization and constraints.

Key takeaway 1: **Heterogeneity is most beneficial when resources are limited, or for larger models, or when the ratio of the different GPU types aligns with their memory and compute characteristics for better load balancing.**

5.2.3 Geo-distributed setups. In Figure 10, we evaluate Sailor's throughput in a setup with 2 cloud regions and 5 zones (4 zones in us-central1 and 1 zone in us-west1), considering A100 GPUs for the OPT-350M model. We compare Sailor with DTFM, with the exhaustive search to automatically discover parallelization plans. We report training throughput and monetary cost per iteration, taking both the computation and communication cost into account.

DTFM cannot fully leverage multiple zones, mainly due to its suboptimal cost function, and lack of memory footprint estimation. DTFM ranks solutions based on the time spent in data and pipeline parallel communication, which leads to suboptimal solution ranking. Sailor achieves 5.9× higher throughput and 9.48× lower cost per iteration. Sailor employs larger microbatch sizes and tensor parallelism degrees, reducing the pipeline and data parallel data transfers. Furthermore, Sailor finds configurations within 1 second, compared to DTFM that needs hundrend of seconds with large clusters (due to the exhaustive search). On GPT-Neo, DTFM fails due to OOMs, while Sailor finds valid plans with a throughput of 0.07–0.21 iters/sec across cluster sizes.

Finally, comparing Sailor's throughput for the OPT-350M model in the heterogeneous setups with more V100 (Figure 8b) and the geo-distributed A100-only setup (Figure 10), shows that the geo-distributed setup achieves up to 2× higher throughput, and also lower cost.

Key takeaway 2: **Efficiently using the same GPU type across zones can lead to higher throughput and lower cost than mixing GPU types within a single zone, despite data transfer costs in geo-distributed setups.**

5.2.4 Optimization with constraints. We now change the optimization objective to minimizing monetary cost and add constraints. Since baselines do not support cost-aware optimization or constraints, we modify them to rank solutions by iteration cost and only return plans within the

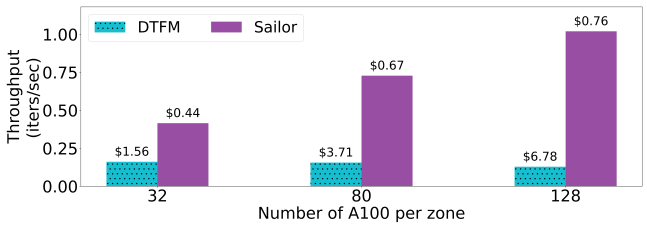


Figure 10. Throughput of geo-distributed planners with A100-40GB for the OPT-350M model in 5 zones (2 regions)

constraints. We consider 2 cloud zones in the same region, each with 128 A100 and 128 V100. Sailor takes the full search space and outputs the resource allocation and parallelization plans. The baselines (that require a fixed topology) assume 4-GPU VMs: Varuna, Aceso, Galvatron consider only the A100 machines (since they are more high-end than V100). AMP, FlashFlex, and Metis consider both A100 and V100 in a single zone, while DTFM considers only A100 in two zones.

Scenario 1: Minimizing cost with throughput constraint of 0.2 iterations/sec: Figure 11 shows the throughput (bars) and iteration cost (asterisks) achieved by the different planners. Sailor outputs a solution within the constraint, while achieving the minimum cost compared to baselines: 40% lower cost compared to the second-best performing baseline (Galvatron). The found solution consisted of 64 A100 GPUs in a single zone, as they were enough to meet the throughput target.

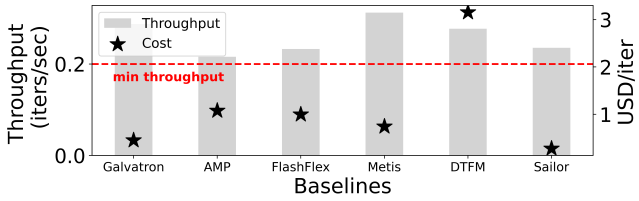


Figure 11. Minimizing cost with a throughput constraint.

Scenario 2: Maximizing throughput with cost constraint of 1.2 USD/iteration: Figure 12 shows the throughput (bars) and iteration cost (asterisks) for this scenario. Most baselines will use all available resources (e.g. all 128 A100 and 128 V100), even if they do not benefit throughput. DTFM does not find a solution as it outputs plans with low throughput and high costs. Sailor outperforms all baselines leading to 1.65-3× higher throughput while remaining within the cost constraint. Sailor's plan includes 256 A100 GPUs in two zones, with tensor parallelism of 4, and data parallelism of 64.

5.3 Sailor scalability study

We evaluated Sailor's search time varying the number of GPUs and zones or regions with a homogeneous GPU type. The search time remains below 1.5 sec even with 5 GCP zones and 256 A100/zone for the GPT-Neo-2.7B model. In

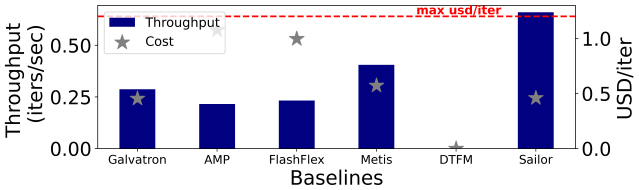


Figure 12. Maximizing throughput with budget limit.

GPU types	Search Time		
	Dyn Prog	+ Heuristics	+ cost limit
1	hours	0.25 sec	0.4 sec
2	hours	5 sec	20 sec

Table 3. Breakdown of search time, in dynamic programming and heuristics, and additional search time overhead due to budget constraints for the GPT-Neo-2.7 model. We use A100 and V100 in one zone, with 128 GPUs per type. The budget constraint is 1.5 USD/iteration.

contrast, adding more GPU types has a much higher impact in Sailor's search time: considering 256 GPUs/type in a single zone, Sailor's search time is 0.3, 6.2, 4900 sec for 1, 2, 3 GPU types, respectively. Nevertheless, Sailor's search process is much more efficient than the rest of the heterogeneous baselines: Metis [56] needs hours to complete its search even with 2 GPU types, while FlashFlex [66] cannot find a valid configuration for any of these setups.

5.4 Sailor Planner optimization breakdown

Table 3 shows Sailor's planner search time breakdown when optimizing for throughput, and budget constraint overhead, considering 128 A100 and 128 V100 GPUs for the GPT-Neo-2.7b model. The dynamic-programming-only approach needs hours to complete the search, even in the single-GPU-type case. Introducing the heuristics *H1-H3* (4.2.1), which apply in this scenario, dramatically decreases the search time to a couple of seconds. The additional cost constraint increases the search time (4× increase in the 2-GPU-type scenario) due to the extra iterations caused by the straggler approximations, as described in 4.2.3.

6 Other Related Work

Asynchronous geo-distributed training: Several systems propose training over geo-distributed, preemptible, and heterogeneous resources by introducing asynchrony and reducing communication via quantization and sparsification. DiLoCo [12] uses federated averaging to reduce communication, performing more local computation before synchronization. SWARM [43] proposes a decentralized, model-parallel approach to deal with poorly connected and unreliable devices. CocktailSGD [59] uses gradient compression to improve communication over low-bandwidth networks. These approaches influence training dynamics and are orthogonal

to our work. Sailor employs synchronous training, which is preferred in large-scale training [57].

Automatic VM selection: CherryPick [2], RAMBO [25], and PARIS [65] apply Bayesian Optimization or performance modeling to recommend optimal VM types. SkyPilot [67] picks cost-efficient VMs across providers based on workload and user constraints. These methods treat workloads as black boxes and are not tailored for ML training. Srifty [29], Cynthia [70], SpotDNN [47], DeepSpot [23] find optimal configurations of on-demand and spot instances for ML jobs. However, they target small data-parallel jobs, and are inadequate for large-scale training that involves various types of communication, which is our focus.

7 Conclusion

We propose Sailor, a system for efficient large-scale training over heterogeneous resources with dynamic availability. Sailor co-optimizes the resource allocation and parallelization plan for a training job to optimize a user-defined objective, under constraints. By combining accurate iteration time and memory estimation, dynamic-programming based search, and domain-specific heuristics, Sailor efficiently navigates the large search space of possible job configurations. Sailor's distributed training framework supports heterogeneous setups and provides seamless elasticity. Sailor achieves 1.1-5.9× higher throughput than baselines across homogeneous, heterogeneous, and geo-distributed settings.

References

[1] Bilge Acun, Benjamin Lee, Fiodar Kazhamiaka, Kiwan Maeng, Udit Gupta, Manoj Chakkaravarthy, David Brooks, and Carole-Jean Wu. Carbon explorer: A holistic framework for designing carbon aware datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 118–132, New York, NY, USA, 2023. Association for Computing Machinery.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, March 2017. USENIX Association.

[3] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 472–487, New York, NY, USA, 2022. Association for Computing Machinery.

[4] AWS. Overview of data transfer costs for common architectures. https://aws.amazon.com/blogs/architecture/overview-of-data-transfer-costs-for-common-architectures/, 2025.

[5] Azure. Azure, bandwidth pricing. https://azure.microsoft.com/en-us/pricing/details/bandwidth/, 2025.

[6] Baseline. China achieves breakthrough in ai training. https://www.baselinemag.com/news/china-achieves-breakthrough-in-ai-training/, 2024.

[7] Runxiang Cheng, Chris Cai, Selman Yilmaz, Rahul Mitra, Malay Bag, Mrinmoy Ghosh, and Tianyin Xu. Towards gpu memory efficiency for distributed training at scale. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 281–297, New York, NY, USA, 2023. Association for Computing Machinery.

[8] Tapan Chugh, Srikanth Kandula, Arvind Krishnamurthy, Ratul Mahajan, and Ishai Menache. Anticipatory resource allocation for ml training. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 410–426, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Google Cloud. Google cloud - about gpus. https://cloud.google.com/compute/docs/gpus/about-gpus, 2024.

[10] Google Cloud. Google cloud, all networking pricing. https://pytorch.org/docs/stable/torch_cuda_memory.html, 2025.

[11] deepspeedai. Megatron-deepspeed. https://github.com/deepspeedai/Megatron-DeepSpeed, 2025.

[12] Arthur Douillard, Qixuan Feng, Andrei A. Rusu, Rachita Chhaparia, Yani Donchev, Adhiguna Kuncoro, Marc'Aurelio Ranzato, Arthur Szlam, and Jiajun Shen. Diloco: Distributed low-communication training of language models, 2024.

[13] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. Parcae: Proactive, Liveput-Optimized DNN training on preemptible instances. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1121–1139, Santa Clara, CA, April 2024. USENIX Association.

[14] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, February 2019. USENIX Association.

[15] Runsheng Benson Guo, Utkarsh Anand, Arthur Chen, and Khuzaima Daudjee. Cephalo: Harnessing heterogeneous gpu clusters for training transformer models, 2024.

[16] Toms Hardware. Meta to build 2gw data center with over 1.3 million nvidia ai gpus, 2025.

[17] 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. *GPipe: efficient training of giant neural networks using pipeline parallelism.* Curran Associates Inc., Red Hook, NY, USA, 2019.

[18] HuggingFace. Huggingface, gpt-neo. https://huggingface.co/docs/transformers/en/model_doc/gpt_neo, 2025.

[19] HuggingFace. Huggingface, opt. https://huggingface.co/docs/transformers/en/model_doc/opt, 2025.

[20] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23. ACM, October 2023.

[21] 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, Renton, WA, July 2019. USENIX Association.

[22] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. Whale: Efficient giant model training over heterogeneous GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 673–688, Carlsbad, CA, July 2022. USENIX Association.

[23] Kyungyong Lee and Myungjun Son. Deepspotcloud: Leveraging cross-region gpu spot instances for deep learning. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 98–105, 2017.

[24] Dacheng Li, Hongyi Wang, Eric Xing, and Hao Zhang. Amp: Automatically finding model parallel strategies with heterogeneity awareness, 2022.

[25] Qian Li, Bin Li, Pietro Mercati, Ramesh Illikkal, Charlie Tai, Michael Kishinevsky, and Christos Kozyrakis. Rambo: Resource allocation for microservices using bayesian optimization. *IEEE Computer Architecture Letters*, 20(1):46–49, 2021.

[26] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, Weijiang Xu, Mao Yang, Lintao Zhang, and Lidong Zhou. nnScaler: Constraint-Guided parallelization plan generation for deep learning training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 347–363, Santa Clara, CA, July 2024. USENIX Association.

[27] Linkedin. The heat challenge of ai infrastructure: A growing concern for traditional office buildings and older data centers. https://www.linkedin.com/pulse/heat-challenge-ai-infrastructure-gpu-servers-trgdatacenter-b6vcc, 2024.

[28] 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*, EuroSys '24, page 163–181, New York, NY, USA, 2024. Association for Computing Machinery.

[29] Liang Luo, Peter West, Pratyush Patel, Arvind Krishnamurthy, and Luis Ceze. Srifty: Swift and thrifty distributed neural network training on the cloud. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 833–847, 2022.

[30] Meta. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation. https://ai.meta.com/blog/llama-4-multimodal-intelligence/, 2025.

[31] Xupeng Miao, Yining Shi, Zhi Yang, Bin Cui, and Zhihao Jia. Sdpipe: A semi-decentralized framework for heterogeneity-aware pipeline-parallel training. *Proc. VLDB Endow.*, 16(9):2354–2363, may 2023.

[32] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *Proc. VLDB Endow.*, 16(3):470–479, 2022.

[33] Zizhao Mo, Huanle Xu, and Chengzhong Xu. Heet: Accelerating elastic training in heterogeneous deep learning clusters. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 499–513, New York, NY, USA, 2024. Association for Computing Machinery.

[34] 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. USENIX Association, February 2021.

[35] 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*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Palak, Rohan Gandhi, Karan Tandon, Debopam Bhattacherjee, and Venkata N. Padmanabhan. Improving training time and gpu utilization in geo-distributed language model training, 2024.

[37] Palak, Rohan Gandhi, Karan Tandon, Debopam Bhattacherjee, and Venkata N. Padmanabhan. Improving training time and gpu utilization in geo-distributed language model training, 2024.

[38] PyTorch. Pytorch cuda events. https://pytorch.org/docs/stable/generated/torch.cuda.Event.html, 2025.

[39] PyTorch. Pytorch hooks. https://pytorch.org/docs/stable/generated/torch.Tensor.register_hook.html, 2025.

[40] PyTorch. Understanding cuda memory usage. https://pytorch.org/docs/stable/torch_cuda_memory.html, 2025.

[41] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Lisa Rivalin, Lingyun Yi, Megan Diefenbach, Alex Bruefach, Frances Amatruda, and Tobias Tiecke. Estimating embodied carbon in data center hardware, down to the individual screws. https://sustainability.atmeta.com/blog/2024/09/10/estimating-embodied-carbon-in-data-center-hardware-down-to-the-individual-screws/, 2025.

[43] Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. Swarm parallelism: Training large models can be surprisingly communication-efficient, 2023.

[44] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[45] Ian Schneider, Hui Xu, Stephan Benecke, David Patterson, Keguo Huang, Parthasarathy Ranganathan, and Cooper Elsworth. Life-cycle emissions of ai hardware: A cradle-to-grave approach and generational trends, 2025.

[46] Semaphor. Microsoft azure cto: Us data centers will soon hit size limits. https://www.semafor.com/article/10/11/2024/microsoft-azure-cto-us-data-centers-will-soon-hit-limits-of-energy-grid, 2024.

[47] Ruitao Shang, Fei Xu, Zhuoyan Bai, Li Chen, Zhi Zhou, and Fangming Liu. spotdnn: Provisioning spot instances for predictable distributed dnn training in the cloud. In *2023 IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)*, pages 1–10, 2023.

[48] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[49] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Esha Choukse, Haoran Qiu, Rodrigo Fonseca, Josep Torrellas, and Ricardo Bianchini. Tapas:

Thermal- and power-aware scheduling for llm inference in cloud platforms. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 1266–1281, New York, NY, USA, 2025. Association for Computing Machinery.

[50] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. Ml training with cloud gpu shortages: Is cross-region the answer? In *Proceedings of the 4th Workshop on Machine Learning and Systems*, EuroMLSys '24, page 107–116, New York, NY, USA, 2024. Association for Computing Machinery.

[51] Foteini Strati, Michal Friedman, and Ana Klimovic. Pccheck: Persistent concurrent checkpointing for ml. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 811–827, New York, NY, USA, 2025. Association for Computing Machinery.

[52] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. Déjàvu: Kv-cache streaming for fast, fault-tolerant generative llm serving, 2024.

[53] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 24829–24840. Curran Associates, Inc., 2021.

[54] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, Boston, MA, April 2023. USENIX Association.

[55] tom's Hardware. Microsoft surprises analysts with massive 80b ai investment plans for 2025, 2025.

[56] 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, Santa Clara, CA, July 2024. USENIX Association.

[57] 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*, SOSP '24, page 195–210, New York, NY, USA, 2024. Association for Computing Machinery.

[58] Jaylen Wang, Daniel S. Berger, Fiodar Kazhamiaka, Celine Irvene, Chaojie Zhang, Esha Choukse, Kali Frost, Rodrigo Fonseca, Brijesh Warrier, Chetan Bansal, Jonathan Stern, Ricardo Bianchini, and Akshitha Sriraman. Designing cloud servers for lower carbon. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 452–470, 2024.

[59] Jue Wang, Yucheng Lu, Binhang Yuan, Beidi Chen, Percy Liang, Christopher De Sa, Christopher Re, and Ce Zhang. Cocktailsgd: fine-tuning foundation models over 500mbps networks. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[60] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, April 2022. USENIX Association.

[61] Wikipedia. Gpt-2. https://en.wikipedia.org/wiki/GPT-4, 2025.

[62] Wikipedia. Gpt-4. https://en.wikipedia.org/wiki/GPT-4, 2025.

[63] Wikipedia. List of nvidia graphics processing units. https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units, 2025.

[64] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. Falcon: Pinpointing and mitigating stragglers for large-scale hybrid-parallel training, 2024.

[65] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. Selecting the best vm across multiple public clouds: a data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 452–465, New York, NY, USA, 2017. Association for Computing Machinery.

[66] Ran Yan, Youhe Jiang, Wangcheng Tao, Xiaonan Nie, Bin Cui, and Binhang Yuan. Flashflex: Accommodating large language model training over heterogeneous environment, 2024.

[67] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, Boston, MA, April 2023. USENIX Association.

[68] Binhang Yuan, Yongjun He, Jared Quincy Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy Liang, Christopher Re, and Ce Zhang. Decentralized training of foundation models in heterogeneous environments, 2023.

[69] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, Santa Clara, CA, July 2017. USENIX Association.

[70] Haoyue Zheng, Fei Xu, Li Chen, Zhi Zhou, and Fangming Liu. Cynthia: Cost-efficient cloud resource provisioning for predictable distributed deep neural network training. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, New York, NY, USA, 2019. Association for Computing Machinery.

[71] 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. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning, 2022.

[72] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Towards optimal large language model serving throughput, 2024.