Zorse: Optimizing LLM Training Efficiency on Heterogeneous GPU Clusters

Runsheng Benson Guo
University of Waterloo
Waterloo, Canada
r9guo@uwaterloo.ca

Utkarsh Anand
University of Waterloo
Waterloo, Canada
utkarsh.anand@uwaterloo.ca

Khuzaima Daudjee
University of Waterloo
Waterloo, Canada
khuzaima.daudjee@uwaterloo.ca

Rathijit Sen
Microsoft
Redmond, USA
rathijit.sen@microsoft.com

Abstract

Large language models (LLMs) require vast amounts of GPU compute to train, but limited availability and high costs of GPUs make homogeneous clusters impractical for many organizations. Instead, assembling heterogeneous clusters by pooling together GPUs of different generations allow them to achieve higher aggregate compute and make use of all available GPUs. However, training on heterogeneous clusters presents several challenges, including load balancing across GPUs, optimizing memory usage to accommodate varying memory capacities, and ensuring communication-efficient training over diverse network interconnects potentially spanning multiple datacenters. In this paper, we make the case that efficient training on heterogeneous clusters requires (1) the integration of pipeline parallelism and data parallelism in a manner that is both communication- and memory-efficient, and (2) a more adaptable configuration of pipeline and data parallelism, which includes the capability to flexibly partition GPUs into asymmetric pipeline parallel stages and to incorporate heterogeneous GPUs within the same data parallelism group. We propose Zorse, the first system to unify all these capabilities while incorporating a planner that automatically configures training strategies for a given workload. Our evaluation shows that Zorse significantly outperforms state-of-the-art systems in heterogeneous training scenarios.

1 Introduction

Large Language Models (LLMs) are built on the transformer architecture and self-attention mechanism [51], which effectively captures contextual dependencies in language data. These models achieve state-of-the-art performance in NLP tasks, including language generation, translation, and summarization, making them highly versatile and widely adopted. LLMs are trained using variants of stochastic gradient descent (SGD) [43], which involves multiple iterations of forward passes, backward passes, and optimizer updates. In the forward pass, a sampled batch of inputs is fed into the model to generate an output. The backward pass then computes the gradients of the model's parameters with respect to the loss, a measure of the discrepancy between the model's outputs and the expected values. In the final step, the optimizer updates the model's parameters using the computed gradients.

The training process involves substantial matrix multiplications, which, although computationally intensive, are highly parallelizable. As a result, training is often accelerated using GPUs, leveraging their thousands of cores for parallel processing. Given the high memory and computational demands of training LLMs, the process is typically distributed across a cluster of GPUs. Distributed training strategies can be broadly categorized into data parallelism and model parallelism. In data parallelism, each GPU processes a separate batch of data using the full model. Techniques like ZeRO-3 [38, 62] reduce memory overhead by sharding training states (parameters, gradients, optimizer state) rather than replicating them across GPUs.

Model parallelism partitions the model itself across multiple GPUs. Pipeline model parallelism [11, 28] partitions a model into stages of consecutive model layers that are assigned to different GPUs. The global batch of inputs is divided into smaller microbatches that are pipelined through the stages to parallelize computation. Rather than dividing the layers of the model, tensor model parallelism [45, 46] divides the computation within each layer across GPUs. This splits the computation at a finer granularity but requires extra communication to aggregate results between layers.

Some systems combine both data and model parallelism to enhance training efficiency [17, 28, 30, 63]. For example, Megatron-LM [30] employs heuristics to select the most appropriate parallelization strategies: tensor parallelism (TP) is used within nodes with fast interconnects, pipeline parallelism (PP) is applied across nodes with slower connections, and data parallelism (DP) is used for further scaling.

Distributed training systems typically assume deployment on clusters of homogeneous GPUs, which simplifies the parallelism configuration. Since each GPU has the same compute and memory capacity, they can be assigned an equal share of the workload. However, many machine learning practitioners do not have access to large, homogeneous clusters due to frequent GPU release cycles, limited cloud availability, and GPU shortages [4, 9, 16, 31, 50, 55, 59]. High-end GPUs are expensive, and most organizations cannot afford to purchase a new cluster every year to keep up with the rapid release of new GPUs. Instead, they often incrementally acquire GPUs, resulting in clusters of heterogeneous GPUs over time. Additionally, several studies have highlighted the limited availability of GPUs in the cloud [5, 9, 16, 48]. Newer GPU models are rarely accessible, and it is often difficult to reserve more than 32 GPUs, even for older generations [9, 16]. Nevertheless, users can still reserve larger GPU clusters for computation by combining GPUs of different generations.

For these reasons, clusters of heterogeneous GPUs are becoming more and more ubiquitous, and there has been lots of interest in developing efficient distributed training techniques for heterogeneous clusters [9, 34, 50, 55, 59]. However, training on such clusters introduces several challenges:
(1) Diverse GPU capabilities: Different GPUs vary in compute power and memory capacity. Hence, efficient training requires flexible parallelism strategies tailored to each device.
(2) Network heterogeneity: Variability in interconnect bandwidth (e.g., inter- vs. intra-node, inter- vs. intra-datacenter) requires communication-efficient parallelism techniques that support large differences in link speeds.
(3) Memory constraints: GPUs with lower memory capacity demand memory-efficient training strategies to maximize compute utilization without running out of memory.

To address these challenges, we introduce Zorse, the first system to resolve all of these issues simultaneously. Zorse combines PP and DP by partitioning the model into stages, pipelining computation across them, and applying DP within each stage. Zorse employs ZeRO-2 DP, which reduces memory consumption by sharding optimizer states and gradients. Unlike existing PP approaches, each stage is divided into ministages, computations are interleaved to avoid fully materializing stage parameters and gradients in memory, Moreover, both activations and parameters are offloaded to CPU memory when not in use, significantly reducing memory overhead. Zorse also supports heterogeneous PP, allowing different stages to use varying numbers of GPUs, and each stage to be parallelized with different GPU configurations. This paper makes the following contributions:

1. We examine key challenges in heterogeneous LLM training and provide insights on designing efficient systems for this use case.
2. We built Zorse, a flexible system that simultaneously addresses these challenges through a novel and efficient integration of pipeline and data parallelism.
3. We design a planner that efficiently navigates the large search space of configurations in Zorse, automatically optimizing training for a given workload and cluster.
4. We demonstrate that Zorse greatly outperforms existing heterogeneous training systems, achieving up to 3x higher training throughput on three representative clusters for models scaling up to 65 billion parameters.

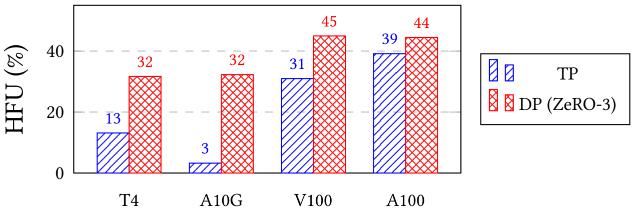


Figure 1. Hardware Flops Utilization (HFU [3]) of tensor parallelism (TP) vs data parallelism (DP) with ZeRO-3 on 8 GPU AWS VMs.

2 Challenges in Heterogeneous Training

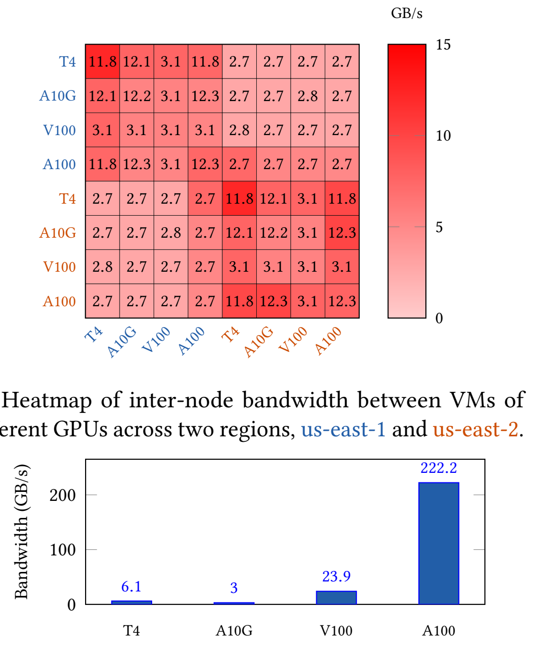
In this section, we provide insights and establish key challenges to overcome for achieving efficient training on heterogeneous GPU clusters.

2.1 Tensor Parallelism in Heterogeneous Clusters

Tensor parallelism (TP) is typically deployed only in clusters of high-end GPU machines interconnected by very high bandwidth technologies such as NVSwitch [30]. This is because TP requires frequent all-to-all communication between model layers. Without extremely high-bandwidth interconnects, the overhead from these communications becomes prohibitively large, despite optimizations that overlap communication with computation [54]. In Figure 1, we compare the GPU utilization when training with TP versus data parallelism (DP) across common AWS VMs. The plot shows that GPU utilization with TP is low relative to DP for most VMs, and is only comparable to DP on the 8×A100 VM that has high bandwidth NVSwitch interconnects. Moreover, TP is generally advantageous only in the following training scenarios where DP or PP become impractical:
(1) Exceptionally large models: TP is useful for training very large models with layers that exceed the GPU's memory capacity, as it partitions the computation within each layer.
(2) Extremely large batch sizes: When DP leads to excessively large batch sizes that impact training convergence, TP can be used to distribute training across more GPUs without further increasing the batch size.

These scenarios, however, are not typical in heterogeneous training environments. Such environments often utilize low- or mid-tier GPUs with limited networking bandwidth, since high-end GPUs are limited in availability [9]. Additionally, organizations using heterogeneous clusters are typically resource constrained, and therefore would not be training at the scale of models and batch sizes that require TP.

Takeaway #1: TP is unsuitable for heterogeneous training due to its high communication overhead and misalignment with heterogeneous training scenarios.



	T4	A10G	V100	A100	T4	A10G	V100	A100
T4	11.8	12.1	3.1	11.8	2.7	2.7	2.7	2.7
A10G	12.1	12.2	3.1	12.3	2.7	2.7	2.8	2.7
V100	3.1	3.1	3.1	3.1	2.8	2.7	2.7	2.7
A100	11.8	12.3	3.1	12.3	2.7	2.7	2.7	2.7
T4	2.7	2.7	2.7	2.7	11.8	12.1	3.1	11.8
A10G	2.7	2.7	2.8	2.7	12.1	12.2	3.1	12.3
V100	2.8	2.7	2.7	2.7	3.1	3.1	3.1	3.1
A100	2.7	2.7	2.7	2.7	11.8	12.3	3.1	12.3

(a) Heatmap of inter-node bandwidth between VMs of different GPUs across two regions, us-east-1 and us-east-2.

(b) Intra-node bandwidth within VMs.

Figure 2. Unidirectional bandwidth between GPUs on AWS.

2.2 Data Parallelism in Heterogeneous Networks

In homogeneous clusters with high-bandwidth interconnects, DP has been shown to scale efficiently to large clusters of over 1000 GPUs [38, 53]. However, the scalability of DP is limited in heterogeneous clusters due to substantial network variability caused by differences in (1) intra-node networking, (2) inter-node networking, and (3) intra- and inter-datacenter networking. Figure 2 illustrates the network heterogeneity in bandwidth between common GPUs/VMs on AWS.

DP uses collective operations like AllReduce, which synchronize model updates across GPUs using all-to-all communication patterns. However, these collectives utilize links inefficiently in heterogeneous networks and can significantly increase communication latencies since faster links are bottlenecked by slower links. Moreover, inter-datacenter bandwidth is typically insufficient to train efficiently with DP [48]. Instead of relying on DP alone, DP can be combined with pipeline parallelism (PP) which requires significantly less communication and can be used over slower inter-node or inter-datacenter links [30, 48].

Takeaway #2: Efficiently scaling training over heterogeneous networks with slow links necessitates a combination of data parallelism and pipeline parallelism.

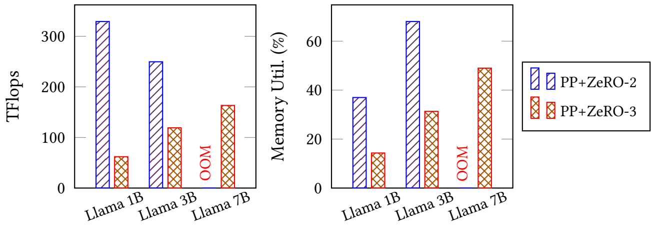


Figure 3. Comparison of pipeline parallelism (PP) with ZeRO-2 vs ZeRO-3 across different Llama[49, 58] model sizes on 8 V100s + 8 T4s. OOM indicates Out-of-Memory.

2.3 PP + DP in Heterogeneous Clusters

ZeRO-3 [38, 53] (or FSDP [62] in PyTorch) is a popular variant of DP designed to avoid redundant storage of training states across GPUs. While this method significantly reduces the memory requirement for storing training states by a factor proportional to the number of GPUs in the cluster, it also increases communication overhead. Specifically, ZeRO-3 must gather parameter shards before each forwards and backwards computation (parameters are sharded afterwards). This overhead is exacerbated when ZeRO-3 is combined with PP, as PP divides a batch of inputs into microbatches that are pipelined to overlap computation across pipeline stages. Consequently, the frequency of parameter gathering increases proportionally with the number of microbatches, creating a significant communication bottleneck and slowing down training unless extremely fast interconnects are available, which is uncommon in heterogeneous clusters.

There have been proposals to amortize the communication cost of gathering parameters in ZeRO-3 by swapping the order of computation [9, 20]. Instead of processing all layers for one microbatch before moving to the next, these approaches sequentially process all microbatches for one layer before proceeding to the next layer. While this strategy reduces the communication overhead when combining ZeRO-3 and PP, it introduces a pipelining overhead: subsequent pipeline stages cannot begin processing until the previous stage reaches the last layer, limiting the ability to overlap computations across stages.

Alternatively, PP can be combined with ZeRO-2, which shards optimizer state and gradients but keeps a full set of parameters in memory. This avoids the need to gather parameters before each computation, achieving higher training throughput but significantly increases memory usage, as shown in Figure 3. This is especially problematic in heterogeneous clusters, where some GPUs can be memory constrained relative to their compute capacity [9]. In Figure 3, although both V100s and T4s have the same memory, V100s are more than three times faster. As model size increases, V100s in PP + ZeRO-2 are unable to process a workload proportional to its compute capacity due to memory constraints, resulting in lower training throughput.

> **Takeaway #3**: Existing variants of DP cannot be combined with PP without introducing significant communication, pipelining, or memory overhead.

2.4 Symmetric Parallelism Configuration

Most state-of-the-art distributed training systems [22, 26, 30, 40] integrate PP with DP by organizing GPUs into a 2D mesh, with PP along one axis and DP on the other. This configuration simplifies system design and narrows the parallelism plan search space. However, in heterogeneous clusters, this approach may overlook efficient training configurations. Such clusters often contain asymmetry, with variations in GPU count per node, hardware, and datacenter location. Efficient training configurations would ideally group GPUs of the same datacenter, node, or GPU type for DP, while applying PP across different groups, ensuring that GPUs within each DP group share similar networking and computational capabilities. Moreover, GPUs may need to be grouped into varying group sizes to accomodate differences in memory capacity (e.g. GPUs with less memory require smaller groups to reduce the memory overhead of parameter replication). Supporting such configurations is important for heterogeneous training, and requires a system capable of asymmetrically partitioning GPUs into PP stages of different sizes.

> **Takeaway #4**: Supporting PP with variably-sized DP stages is essential for optimizing training on heterogeneous clusters.

2.5 Pipelining Overheads

Training systems optimized for homogeneous clusters [22, 26, 30, 40, 63] typically distribute batch sizes evenly across GPUs within a DP stage. This uniform allocation results in training bottlenecks when heterogeneous GPUs share a stage, as faster GPUs will be blocked waiting for slower GPUs within the same stage. Avoiding this issue in heterogeneous environments restricts GPUs within a DP group to be of the same type, which therefore scales the number of PP stages by the number of GPU types. However, adding more pipeline stages can introduce inefficiencies, such as increased idle time (bubbles) and additional inter-stage communication. Moreover, increasing the number of stages constrains the flexibility in distributing model layers across stages to achieve balanced computation, especially considering the relatively few layers in modern LLMs. Consequently, it is important for heterogeneous training systems to support balancing of computational loads across different GPUs within a stage. This capability allows for configurations with fewer stages that incur lower pipelining overhead.

> **Takeaway #5**: Balancing computational loads across heterogeneous GPUs within a DP stage facilitates training configurations with reduced pipelining overhead.

3 Limitations of Existing Systems

Given the challenges discussed above, we examine the limitations of representative systems that embody state-of-the-art methods for heterogeneous training.

General 3D Parallel Frameworks. Systems such as TorchTitan [22] and DeepSpeed [40] support 3D parallelism with DP, PP, and TP, but are not explicitly designed for heterogeneous clusters. In heterogeneous clusters, these frameworks need to be manually configured with asymmetric model partitions to balance computation time across stages in PP. However, they do not provide the flexibility to asymmetrically partition GPUs across pipeline stages or to use different batch sizes within DP groups to accommodate heterogeneous GPUs within a stage. Additionally, its DP can be optimized for either memory or communication efficiency, but not both simultaneously. For instance, in TorchTitan, DP can configured to use either ZeRO-2 which is communication efficient but not memory efficient, or ZeRO-3 which is memory efficient but not communication efficient.

Heterogeneity-Aware 3D Parallel Frameworks. Frameworks such as HexiScale [55] and Metis [50] are designed for 3D parallelism in heterogeneous clusters. They support asymmetric PP and permit varying batch sizes within a DP group, enabling balanced computation across different GPUs. However, some parameters still require manual tuning, e.g., HexiScale requires specifying the batch size distribution for DP and microbatches for PP. Additionally, these frameworks do not support ZeRO-3 due to communication bottlenecks when integrated with PP. Instead, they employ ZeRO-2 and standard DP, respectively. When PP and DP exceed the cluster's memory capacity, TP is employed to reduce memory usage, but this often leads to poor utilization in heterogeneous clusters (Section 2.1).

Heterogeneity-Aware ZeRO-3. Systems like Cephalo [9] and Poplar [60] utilize ZeRO-3 for training on heterogeneous clusters. Instead of using PP or TP to scale training for larger batch sizes, Cephalo employs gradient accumulation to reduce the memory requirements by splitting large batch sizes into smaller microbatches. It also reorders the computation to avoid regathering parameters for each microbatch. However, due to the lack of support for PP, Cephalo cannot efficiently utilize network resources in highly heterogeneous networks, resulting in networking bottlenecks when the batch size is insufficient to hide communication latencies [9].

System	Communication Efficient DP	Memory Efficient DP	Computation Balancing DP	PP +DP	Asymmetric PP
TorchTitan-ZeRO2 [22]	✓	×	×	(✓)	×
TorchTitan-ZeRO3 [22]	×	✓	×	(✓)	×
HexiScale [55]	✓	×	(✓)	(✓)	✓
Cephalo [9]	✓	✓	✓	×	×
Zorse (Our System)	✓	✓	✓	✓	✓

✓ Supported, (✓) Supported with manual tuning, × Not supported

Table 1. Comparison of heterogeneous training systems.

4 Design

To overcome the challenges in heterogeneous training discussed in Section 2 and the limitations of existing systems in Section 3, we developed Zorse. Table 1 compares Zorse with existing systems. Zorse is the first to integrate all essential capabilities for efficient training on heterogeneous clusters.

Zorse trains LLMs with a combination of PP and ZeRO-2 DP, using a combination of interleaved pipelining and offloading to optimize the memory efficiency of ZeRO-2 DP while avoiding the communication overhead of ZeRO-3 DP. Zorse also supports flexibly configuring PP and DP, including asymmetric partitioning of GPUs across pipeline stages and varying batch sizes within a DP group. Finally, Zorse's planner automatically determines optimized training configurations for a given workload. In this section, we describe the components of Zorse in detail.

4.1 Efficient PP and DP Integration

Instead of conventional PP which assigns one stage per GPU, Zorse uses interleaved pipelining which assigns multiple smaller *ministages* to each GPU and interleaves computation across them. However, traditional interleaved pipelining [30] uses a 1F1B [28] schedule, interleaving computation across ministages. This means that ministage parameters cannot be sharded without incurring extra communication overhead.

4.1.1 Interleaved Pipelining. Zorse instead uses a GPipe-style [11] pipelining schedule to efficiently integrate ZeRO-2 DP with PP, illustrated in Figure 4. The forwards pass is processed for each ministage sequentially, finishing all microbatches for a ministage before moving on to the next. The backwards pass is processed similarly in the reverse order. Since the microbatches are all processed together for each ministage, we offload other ministages to CPU memory when they are not being used. With this design, Zorse needs to maintain only the parameters for 2 ministages in memory at a time (the current ministage and the next ministage that is being prefetched in parallel), rather than all the parameters in the stage in standard PP + ZeRO-2.

Let L be the total number of layers in the stage, S be the number of stages, P_{layer} be the number of parameters in each layer, D_{dp} be the degree of DP, and M be the number of microbatches. Table 2 summarizes the parameter memory and communication requirements of Zorse, PP + ZeRO-2, and PP + ZeRO-3. As the number of ministages increases, the memory utilization of Zorse approaches that of PP + ZeRO-3 and is significantly lower than PP + ZeRO-2. Additionally, Zorse has the same communication requirement as PP + ZeRO-2, with one AllGather per layer in the forwards and backwards pass. This is significantly lower than PP + ZeRO-3, which requires one AllGather per microbatch of each layer.

Strategy	Materialized Parameters	Sharded Parameters	# AllGathers
Zorse	$2 \times \frac{L}{S} \times P_{layer}$	0	$2 \times L$
PP + ZeRO-2	$L \times P_{layer}$	0	$2 \times L$
PP + ZeRO-3	$2 \times P_{layer}$	$(L-2) \times \frac{P_{layer}}{D_{dp}}$	$2 \times L \times M$

Table 2. Memory and communication comparison of different training strategies.

4.1.2 Interleaved Optimizer Updates. Zorse further takes advantage of its interleaved pipelining schedule to interleave optimizer updates. Instead of starting the optimizer update after the backwards pass completes for all layers, Zorse starts the optimizer update for each ministage as soon as its own backwards pass completes, while the computation for the next ministage is simultaneously starting. This design (Figure 5) offers two key advantages:

(1) Reduced peak memory utilization: By freeing the gradients of each ministage as soon as its optimizer update is finished, Zorse avoids the need to retain gradients until the entire backward pass is complete.

(2) Decreased communication overhead: The optimizer update requires averaging gradients across GPUs within its DP group. By interleaving optimizer updates, this communication can overlap with the computation of subsequent ministages. This overlap is not possible if all optimizer updates start at the end of the pipeline schedule.

4.1.3 Activation Checkpoining & Offloading. As in prior work [30, 36, 55], Zorse checkpoints activations after each transformer layer, discarding activations between layer boundaries and recomputing them when they are needed in the backwards pass. However, the remaining activations at layer boundaries still require a significant chunk of GPU memory due to the pipeline schedule. Sequential processing of all forwards passes before backwards passes means that layer boundary activations for all microbatches of all layers need to be maintained in memory. With a batch size of B, L layers, sequences of length S, a hidden layer size of H, and D bytes per parameter, the memory overhead is $B \times L$ activations of size $S \times H \times D$ bytes. Even at smaller training scales, this can amount to many GBs of memory.

Thus, in addition to offloading parameters, Zorse also offloads layer boundary activations to CPU memory during the forwards pass, which are loaded back during the backwards pass. With offloading, Zorse needs to maintain activations

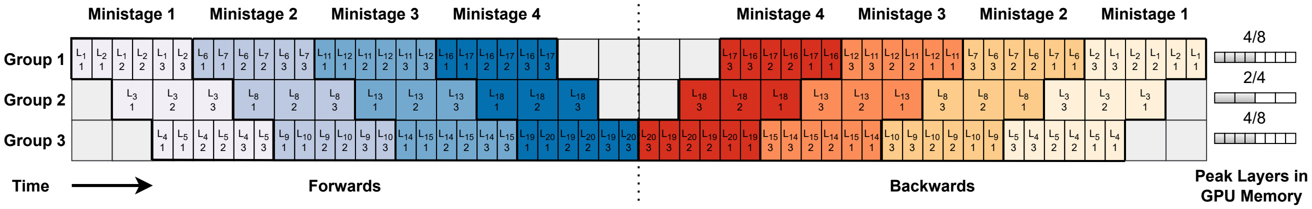


Figure 4. Interleaved pipelining in Zorse. The diagram shows a training iteration for a model with 20 layers (L_i) and 3 microbatches. There are 3 GPU groups, each with 4 ministages. Due to different computational speeds, groups have different numbers of layers per ministage. Each group maintains the current and next ministage, offloading others to CPU memory.

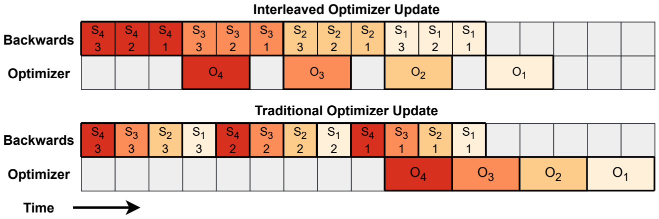


Figure 5. Interleaved optimizer updates in Zorse vs. traditional pipeline parallelism (4 ministages, 3 microbatches). Interleaved optimizer updates free gradient memory earlier and better overlap with computation.

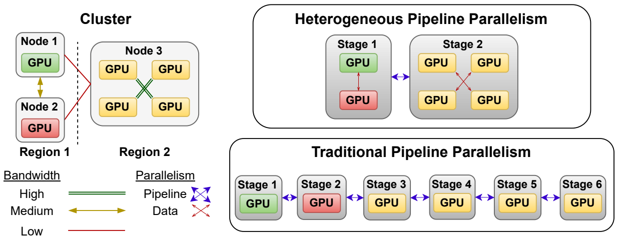


Figure 6. Heterogeneous pipeline parallelism in Zorse compared to traditional pipeline parallelism.

for only the current microbatch being computed and for the next microbatch being prefetched.

In Section 5.4, we describe how Zorse efficiently overlaps offloading (and loading) of activations, and model parameters with computation to hide offloading overhead.

4.2 Heterogeneous Pipeline Parallelism

Zorse supports *heterogeneous pipeline parallelism*, which allows for each pipeline stage to have: (1) a different number of GPUs, and (2) a combination of different GPU types.

This differs from traditional PP, which fixes a uniform number of GPUs per stage and the same GPU type within each stage. This flexibility allows Zorse to more efficiently configure PP with DP in heterogeneous clusters, aligning with the varying numbers of each GPU type, varying numbers of GPUs per VM, and varying memory and compute capabilities in heterogeneous clusters.

Figure 6 illustrates heterogeneous PP on a 3-node cluster with 1, 2, and 4 GPUs of different types across two regions. With heterogeneous PP, nodes in Region 1 form a single stage with 3 GPUs, while Region 2's node forms another stage with 4 GPUs. This approach leverages high-bandwidth intra-region connections for DP communication while using PP across the lower-bandwidth inter-region link, reducing pipeline stages to just two. Traditional PP, however, requires six single-GPU stages to maintain GPU uniformity, significantly increasing pipeline overhead. Moreover, with more stages, it may become infeasible to partition the model layers in a way that evenly distributes computation across stages. This is especially true for LLM models, which have a limited number of layers.

Cross-stage Communication Algorithm. In traditional PP, communication is straightforward due to the uniformity of GPUs both within and across stages, allowing for one-to-one communication between GPUs in different stages. However, in heterogeneous PP where the number and type of GPUs per stage varies, data must be reshuffled across stages, necessitating many-to-many communication patterns. Additionally, data must be balanced across heterogeneous GPUs within a stage to evenly distribute computation.

Zorse computes an optimized communication plan to redistribute microbatches across stages and balance computation within stages. Microbatches are assigned to GPUs based on their relative layer runtimes. This algorithm estimates completion times for microbatches of the current stage and remaining compute time for each GPU in the next stage. It then assigns the ith completed microbatch to the GPU with the ith highest remaining runtime, prioritizing GPUs with more work to minimize overall stage runtime.

4.3 Planner

Zorse initially profiles the cluster and workload to gather model runtime and networking statistics. This data is then utilized by the planner to determine the final training configuration. Zorse's planner employs a two-phase optimization process to optimize the training configuration:

- In Phase 1, the cluster is partitioned into *GPU groups*. PP will be applied across GPU groups and DP within

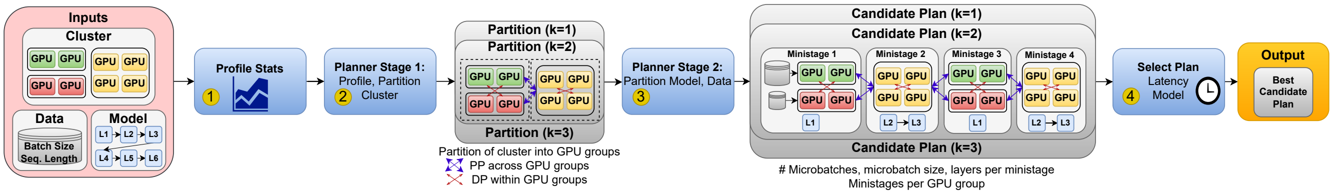


Figure 7. Architecture of Zorse's planner: ① profile workload and cluster, ② partition cluster into GPU groups, ③ partition model and data across GPU groups, and ④ select the best plan considered.

each GPU group. The planner considers the best way to partition the cluster into k GPU groups, for $k \in [1, N]$ where N is the number of GPUs in the cluster.

- In Phase 2, the planner computes an optimized configuration for each cluster partition from Phase 1. This includes the number of microbatches, the size of each microbatch, the number of ministages per GPU group, and the partitioning of the model into these ministages.

The architecture of Zorse's planner is shown in Figure 7.

4.3.1 Profiling.

We measure inter-node and intra-node bandwidths between pairs of nodes and GPUs, respectively, for use in cluster partitioning. We profile model layer runtimes on each GPU for small batch sizes and fit a linear model to predict runtimes unseen batch sizes. These bandwidth measurements and runtime predictions are used to estimate training latencies in Phase 2. To accelerate profiling, we: (1) parallelize intra-node profiling, and (2) profile inter-node bandwidths only between unique VM configurations, as large clusters often contain redundant configurations.

4.3.2 Phase 1: Cluster Partitioning.

In the first phase, the planner determines how to divide the cluster nodes into k GPU groups. GPUs within the same group utilize DP, while PP is applied across groups. Therefore, the goal is to create a partition that minimizes inter-group bandwidth usage. This results in an efficient setup that uses communication-efficient PP over lower bandwidth links between groups and DP over higher bandwidth links within groups.

We represent the cluster as a fully connected graph $G = (V, E)$, where V represents the GPUs and E represents the edges with weights $w(u, v)$ indicating the bandwidth between GPUs u and v. The partitioning task is framed as a min-k cut problem on this graph. Specifically, the objective is to divide V into k disjoint subsets $V_1, V_2, \ldots, V_k$ such that the total weight of the edges between different subsets is minimized. This is called the *min-k cut*:

$$\min \sum_{\substack{u \in V_i, v \in V_j \\ i \neq j}} w(u, v), \quad \text{where} \bigcup_{i=1}^{k} V_i = V,\ V_i \cap V_j = \emptyset \text{ if } i \neq j\,.$$

This partitioning minimizes the bandwidth sum between GPU groups, ensuring that higher bandwidth links are used within groups. This problem is solved for each k from 1 to the total number of GPUs in the cluster, N.

Since exact min-k cut solutions have an impractically large $O(N^{k^2})$ time complexity [8], we use the SPLIT greedy approximation algorithm [44], which guarantees a solution within a factor of $2 - 2/k$ of optimal. SPLIT iteratively computes min 2-cuts of the current graph and removes those edges until k connected components remain. Thus, we can efficiently generate approximate min k cuts for all values from 2 to N in a single execution with $k = N$. With each min 2-cut requiring $O(N^3)$ time [47] and N iterations needed, the overall time complexity of cluster partitioning is $O(N^4)$.

4.3.3 Phase 2: Model Configuration.

In the second phase, the planner utilizes cluster partitions from Phase 1 to determine an optimized training configuration, which encompasses model layer partitioning into ministages, GPU group ordering, and microbatch size selection. Due to the extensive search space, Zorse employs heuristics to prune candidate configurations before evaluating the remaining options with a latency model for performance assessment and a memory model to verify GPU memory constraints are respected.

Heuristics. In pipeline parallelism, the pipeline is bottlenecked by the slowest GPU group. Thus, to minimize training latency, we need to balance runtime across GPU groups by partitioning model layers proportionally to each group's aggregate processing speed, estimated as the sum of layer processing rates across GPUs in the group. Layers assigned to each GPU group are divided into evenly sized ministages, which are then ordered across groups in round robin fashion.

The order of GPU groups impacts pipeline startup time because the first ministage must gather sharded model parameters without having prior ministages to overlap this communication. We order GPU groups by descending intra-group bandwidth to minimize initial communication delay and expedite pipeline startup. This arrangement enables subsequent GPU groups to better overlap their communication with computation from preceding GPU groups.

Enumeration. We enumerate over all possible configurations of batch size and number of ministages per GPU group, which is at most the number of model layers. Hence, there are only $O(B \cdot L)$ configurations to consider, where B is the global batch size and L the number of layers. We use our lightweight latency and memory model to select the configuration with the lowest estimated latency that satisfies the memory constraints.

4.3.4 Latency Model. We model the total training latency as:

$$L_{total} = (L_{forwards} + L_{backwards}) \cdot N_{ministages} + L_{startup} \quad (1)$$

Ministage forward pass latency $L_{forwards}$ encompasses computation time across all GPUs, AllGather communication, PP communication, and accounts for their overlap. Ministage backward pass latency $L_{backwards}$ similarly accounts for computation time (with activation recomputation) and communication overhead for AllGather and ReduceScatter operations. Pipeline startup latency $L_{startup}$ represents the time needed to initialize the pipeline, including initial parameter AllGather and PP communication latencies that cannot overlap with computation.

4.3.5 Memory Model. We model the total per-GPU memory consumption as:

$$M_{total} = M_{params} + M_{grads} + M_{optim} + M_{activations} \quad (2)$$

where M_{params} represents memory for model parameters, M_{grads} gradients, M_{optim} optimizer states, and $M_{activations}$ activations.

5 Implementation

In this section we provide more details on Zorse's implementation and optimizations. Zorse is built on top of FSDP [62] (ZeRO-2 and ZeRO-3 in PyTorch). We integrated interleaved pipeline parallelism with FSDP, handling management of ministages, pipelining microbatches across GPU ministages, and communication between GPU groups.

5.1 Interleaved Pipelining

We modified FSDP's parameter management to support interleaved pipeline parallelism by addressing key timing assumptions. Unlike FSDP's ZeRO-2 which keeps parameters materialized between forward and backward passes, we reshard and offload ministage parameters to CPU post-forward pass, then reload and unshard during the backward passes. To handle FSDP's premature resharding after the first microbatch's backward pass, our implementation delays resharding until all microbatches complete. We limit memory usage by configuring the FSDP prefetcher to fetch only the next immediate ministage. Finally, we create per-ministage optimizers that execute independently after their corresponding backward passes complete.

5.2 Communication Libraries

We use NCCL [32] for DP communication within GPU groups. For PP communication, we encountered limitations with NCCL's blocking P2P behavior [33], which can cause deadlocks in heterogeneous PP due to cyclic dependencies. We implemented a hybrid approach using NCCL where possible and GLOO P2P (non-blocking) between GPU groups where cycles in communication are possible. While GLOO cannot utilize NVLink, this was not a performance bottleneck as NVLink is rare in heterogeneous clusters and typically connects GPUs within the same DP group rather than across PP groups.

5.3 Computation-Networking Overlap

We optimize communication-computation overlap by managing parameter sharding at the layer level instead of ministage level. By gathering layers sequentially rather than as a single large chunk, computation can begin once the first layer is ready while subsequent layers are prefetched in parallel. This approach extends to the backward pass, where we prefetch CPU-offloaded parameters ahead of when they are needed, preventing computation stalls.

5.4 Computation-Offloading Overlap

PyTorch's built-in CPU offloading incurs significant performance overhead by blocking GPU computation and executing optimizer steps on CPU when parameter offloading is enabled. Our custom implementation utilizes separate CUDA streams for parameter and activation offloading without blocking GPU computation. Similarly, we prefetch activations and parameters in advance to prevent stalling during backward pass. Finally, we execute optimizer steps on GPU prior to offloading to avoid costly optimizer updates on CPU.

6 Performance Evaluation

We compare Zorse to state-of-the-art heterogeneous training systems across three representative clusters with up to 128 GPUs, and LLMs with up to 65B parameters. Training throughput and cluster utilization is evaluated in Section 6.2 and 6.3. We evaluate Zorse's training efficiency on heterogeneous clusters in Section 6.4, and investigate how components of Zorse such as offloading and interleaved pipelining contribute to its performance in Section 6.5 and 6.6. Finally, we break down the optimizer runtime in Section 6.7.

6.1 Experimental Setup

For all systems, we train with a global batch size of 1 million tokens using FP16 mixed precision and apply activation checkpointing to reduce memory overhead [36, 55]. We use standard Llama [49] models with 7 to 65 billion parameters.
Clusters. We evaluate on three clusters representative of common heterogeneous training scenarios. These clusters include VMs from Azure and AWS, contain up to 128 GPUs of low-, mid-, and high-end GPUs, and span up to 2 regions. Table 3 details GPU specifications, while Table 4 presents the cluster configurations. We use sequence lengths of 4096, 1024, and 512 for clusters A, B, and C respectively, training with longer sequences on clusters with more powerful GPUs.

Table 3. Datacenter-class GPU Specifications

Performance	GPU	Memory	TFlops (FP16)
High	H100-NVL	94 GB	989
	A100	40/80 GB	312
Middle	V100	16 GB	125
	A10G	24 GB	125
Low	T4	16 GB	65

Table 4. Cluster GPU Configurations

Cluster	Cloud	# VMs	GPUs / VM	Total Regions	Total GPUs	Total TFlops (FP16)
A	Azure	2	2×H100	1	20	8332
		2	8×A100 (80GB)			
B	AWS	1	8×A100 (40GB)	1	64	8112
		2	8×A10G			
		2	8×V100			
		3	8×T4			
C	AWS	2	8×A10G	2	128	8240
		2	8×V100			
		12	8×T4			

Baselines. We compare against representative state-of-the-art techniques for training on heterogeneous GPU clusters:

- TorchTitan-Het: 3D parallelism in TorchTitan [22] adapted for heterogeneous clusters by partitioning the model unevenly to balance compute across PP stages. We try different combinations of DP (ZeRO-2), PP, and TP, reporting the best performing configuration.
- HexiScale [55]: Combines DP (ZeRO-2), tensor, and pipeline parallelism, leveraging a planner to select an optimized training configuration.
- Cephalo [9]: Distributes compute workload and training state unevenly with FSDP to utilize heterogeneous resources efficiently.

Metrics. We evaluate training performance using two key metrics: (1) TFlops: The floating point operations per second achieved during training, which measures training throughput. (2) Hardware FLOPS Utilization (HFU [3]): The ratio of achieved TFlops to the cluster's peak theoretical TFlops, which quantifies GPU utilization efficiency.

6.2 Training Throughput

In this section we compare Zorse to existing systems across three representative heterogeneous training scenarios. A summary of the results is provided in Table 5. More details on the training configurations

A: Small-size cluster of high-end GPUs. This cluster consists of 4 H100s and 16 A100s, representative of scenarios where users have limited high-end GPUs but can combine them to form a larger cluster. GPUs within each node are connected via NVSwitch, while inter-node bandwidth is significantly lower at 50 Gbps. Zorse achieves superior performance over all baselines, with its relative speedup increasing with model size, reaching up to 3× on the LLama 65B model.

TorchTitan-Het partitions the model into 5 stages, each with 4 GPUs, one grouping the 4 H100s together and the rest grouping the A100s. It employs a 2 × 2 ZeRO-2 DP×TP configuration, which works well for smaller models but runs out of memory on larger ones. Gradient accumulation is used to manage memory constraints, but this adds communication overhead for ZeRO-2 from extra parameter gathering.

HexiScale employs PP + ZeRO-2 DP, adjusting PP and DP degrees to manage memory usage. However, larger models still face high memory pressure, leading to suboptimal partitioning and underutilization of H100 GPUs. Despite having nearly 3× the TFlops of A100s, H100s have only 15% more memory, and are unable to train with 3× the number of layers without running out of memory.

Cephalo uses ZeRO-3 across all GPUs, but the significant disparity in intra-node and inter-node bandwidths (over 35-fold) causes AllGather and ReduceScatter communications to bottleneck training.

In contrast, Zorse maintains high efficiency even with larger models by grouping GPUs within each node for DP and applying PP across nodes, possible with Zorse's support for heterogeneous PP. Its memory optimizations allow it to train larger models while evenly balancing computation across GPUs, unlike other systems that must compromise performance to avoid running out of memory.

B: Medium-size cluster of low, middle, & high end GPUs. This cluster comprises 8 A100s, 16 A10Gs, 16 V100s, and 24 T4s, representing scenarios where users have limited access to GPUs with diverse performance capabilities but aim to utilize them concurrently for training. The cluster presents challenges due to its heterogeneity in computational power, memory, and networking. For instance, V100s and A100s benefit from faster intra-node NVLink interconnects, whereas T4s and A10Gs rely on PCIe; T4s and V100s share the same memory capacity, yet V100s are twice as fast; A10Gs and V100s have similar computational speeds, but A10Gs offer 1.5 times more memory.

Zorse partitions the cluster into four groups, each consisting of VMs with the same GPUs and hardware. This setup facilitates ZeRO-2 DP within groups sharing the same networking hardware, ensuring slower interconnects do not impede faster ones. For the larger 13B and 33B models, Zorse employs 4 and 6 ministages per GPU, respectively, increasing parameter offloading and reducing memory usage. This strategy is crucial for performance, as it frees memory, allowing for more flexible layer distribution across GPUs. Zorse consistently achieves a 1.5× – 4× speedup in training throughput across all models and baselines.

TorchTitan-Het and HexiScale lack efficient activation offloading mechanisms, relying instead on partitioning models into many stages with PP to manage memory usage. However, without interleaved pipelining, they struggle to balance computation across PP stages due to the limited number of model layers. For example, FlashFlex divides the LLama 33B model into over 16 stages, but with only 40 layers available, the partitioning is too coarse to balance computation effectively. The A100, with 5× the TFlops of the T4, would need

Table 5. Throughput (TFlops) and GPU utilization (HFU) of Zorse compared to other systems across different model sizes and clusters (higher is better). *OOM* denotes Out-of-Memory.

Cluster	**Model**	**Zorse**		**TorchTitan-Het**		**HexiScale**		**Cephalo**	
		TFlops	**HFU**	**TFlops**	**HFU**	**TFlops**	**HFU**	**TFlops**	**HFU**
A	Llama 7B	**4370.56**	**52.46%**	4223.80	50.69%	3193.46	38.33%	1714.52	20.58%
	Llama 13B	**4917.87**	**59.02%**	3837.49	46.06%	3270.32	39.25%	1656.29	19.88%
	Llama 33B	**5281.64**	**63.39%**	944.47	11.34%	3064.22	36.78%	1943.89	23.33%
	Llama 65B	**5239.13**	**62.88%**	*OOM*	*OOM*	2048.63	24.59%	1937.64	23.26%
B	Llama 7B	**3412.88**	**43.49%**	2033.53	25.91%	1194.89	15.23%	2274.50	28.98%
	Llama 13B	**2965.64**	**37.79%**	1956.09	24.93%	1152.73	14.69%	1992.24	25.39%
	Llama 33B	**2658.29**	**33.87%**	*OOM*	*OOM*	657.16	8.37%	1373.31	17.50%
C	Llama 7B	**3936.94**	**39.24%**	2441.70	24.34%	2624.63	26.16%	1213.39	12.10%
	Llama 13B	**3357.97**	**33.47%**	2061.55	20.55%	1952.31	19.46%	1222.96	12.19%
	Llama 33B	**1548.60**	**15.44%**	*OOM*	*OOM*	*OOM*	*OOM*	775.42	7.73%

5× more layers to achieve balance. Unlike Zorse, TorchTitan-Het cannot flexibly configure PP and DP, and runs out of memory when training LLama 33B.

Cephalo efficiently manages memory by fully sharding parameters across the cluster, balancing computation across GPUs. However, it faces bottlenecks due to collective communication in the highly heterogeneous network.

C: Large-size cluster of low- & middle-end GPUs. This cluster comprises 128 GPUs across two AWS regions: 16 A10Gs and 48 T4s in one, and 16 V100s and 48 T4s in the other. It represents scenarios where users lack high-end GPUs but have access to numerous low- and mid-tier GPUs. Training is challenging due to limited GPU memory (≤ 24GB) and high communication latencies from the large GPU count and slower cross-region links. Consequently, training throughput and utilization are generally lower. Nevertheless, Zorse consistently achieves at least a 1.5× speedup over baselines.

To prevent OOM errors on larger models, Zorse employs varied DP sizes tailored to each VM configuration. For instance, all A10Gs form a single DP group, while V100s are divided into two groups of 8 GPUs each, as they need to handle a similar layer count as A10Gs but with 50% less memory. This smaller DP grouping reduces memory demands for model parameter replication. T4s, assigned fewer layers due to their slower speed, are split into two groups of 24 within each region to avoid cross-region DP.

TorchTitan-Het, unable to asymmetrically partition GPUs, required a higher PP degree to manage memory, leading to high pipelining overhead and poor performance. HexiScale's use of TP to cut memory usage resulted in high communication overhead and also led to poor performance. Cephalo's uneven training state partitioning allowed V100s to store less state, but the cross-region network's uneven partitioning overhead caused communication bottlenecks.

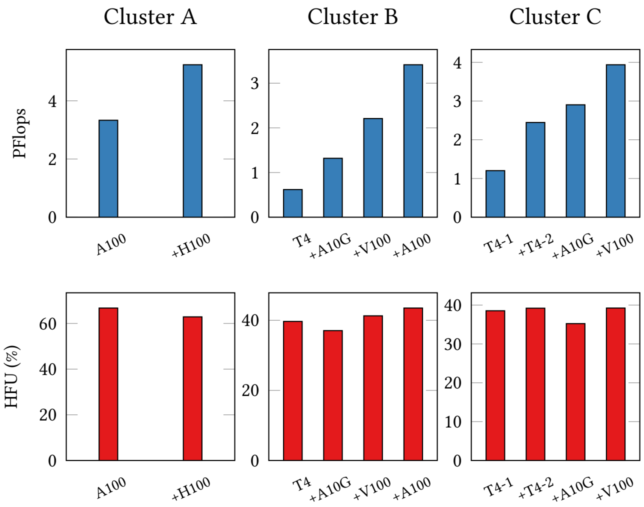


Figure 8. PFlops (1k TFlops) and HFU scaling as heterogeneous GPUs are added to the clusters.

6.3 Cluster Scaling

We validate Zorse's scalability with increasingly heterogeneous and larger clusters. For each cluster, we first evaluate the training performance of the largest model that can be trained on the slowest GPUs. We then incrementally add faster GPUs to the training group, gradually increasing the total number of GPUs and heterogeneity until the entire cluster is utilized. Results are presented in Figure 8.

Adding heterogeneous GPUs into the training group across various clusters significantly improves training throughput. Furthermore, cluster utilization (HFU) generally remains stable or improves, as observed in Cluster B. Improvements can be attributed to reduced memory requirements per GPU with the addition of more GPUs, enabling more efficient training configurations that better balance computation. These results demonstrate that when homogeneous GPUs are limited, they can be pooled together to achieve higher throughput without sacrificing training efficiency with Zorse.

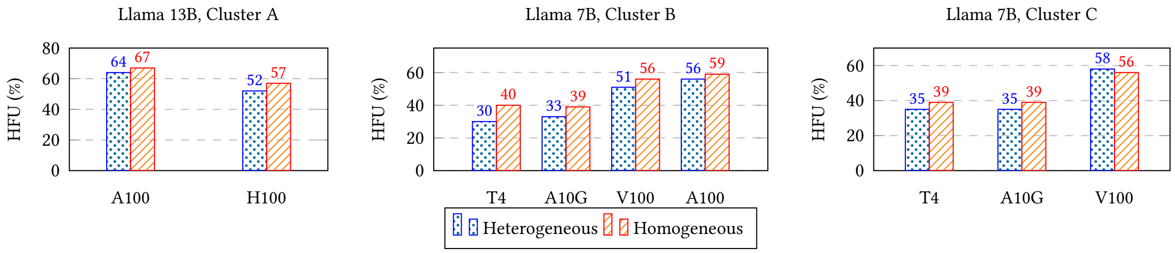


Figure 9. GPU Utilization on heterogeneous clusters vs homogeneous subgroups of GPUs within the clusters.

6.4 Comparison to Homogeneous Training

We assess the training efficiency of Zorse on heterogeneous versus homogeneous clusters in Figure 9. For each heterogeneous cluster, we compare the HFU achieved per GPU type when training on the entire cluster against training on only the homogeneous subset of those GPUs. This evaluation uses the largest model that can be trained on the clusters without running out of memory. Homogeneous training is typically expected be more efficient due to its smaller scale and uniform hardware, which reduces communication overhead and eliminates the need for balancing computation. Nevertheless, our results demonstrate that Zorse consistently achieves efficiency levels comparable to homogeneous training, reflecting its ability to balance network, compute, and memory heterogeneity. This highlights Zorse's capability to effectively scale training across heterogeneous clusters, achieving higher throughput than possible with homogeneous clusters, while maintaining efficiency.

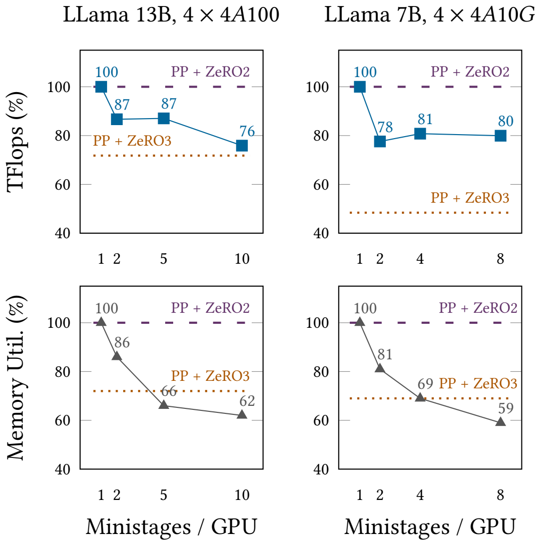


Figure 10. TFlops and memory utilization for varying ministages per GPU. Values are normalized to 1 microstage.

6.5 Interleaved Pipelining Analysis

In Section 2.3, we demonstrated how PP + ZeRO-3 is memory efficient but can lead to very low throughput due to high communication overhead, whereas PP + ZeRO-2 is memory inefficient but can achieve high throughput due to low communication overhead. We analyze how interleaved pipelining in Zorse is able to achieve a good balance between low memory and high throughput, and show the tradeoff as we scale the interleaving factor, corresponding to the number of ministages assigned to each GPU.

In Figure 10, we evaluate the effects of interleaving using two homogeneous clusters of 16 A100s, and 16 A10Gs. We use a homogeneous cluster to isolate the impact of interleaving from side effects that may result from compute imbalance in heterogeneous clusters. There is a slight drop in throughput when going from 1 ministage (no interleaving) to 2 ministages per GPU, due to increases in pipelining overhead. However, as we increase the number of ministages past 2, the additional drop in ministages is minimal, while continuing to decrease memory utilization. In both training setups, with maximum interleaving, we are able to reduce the memory utilization by 40% while incurring only a 20% drop in throughput. When memory is limited in heterogeneous clusters, this is an effective tradeoff since it allows for compute to be balanced more evenly across GPUs, which can often improve throughput and make up for the additional pipelining overhead.

We also plot the performance of PP with ZeRO-2 and ZeRO-3 for comparison. Zorse achieves much higher throughput than PP + ZeRO-3 with comparable memory utilization, particularly on the A10Gs (due to slower links). When interleaving is maximized, Zorse actually uses less memory since it also offloads sharded parameters. We measured offloading overhead to be minimal, contributing at most 3% performance impact across all workloads.

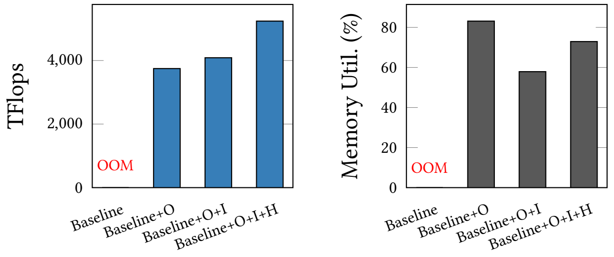


Figure 11. TFlops and memory utilization of Zorse optimizations (Cluster A, LLama 65B). Baseline runs out of memory.

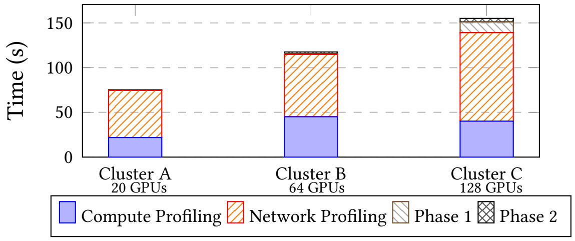


Figure 12. Planner runtime breakdown per cluster.

6.6 Ablation Study

We conduct an ablation study to evaluate the impact of Zorse's key components on training throughput and memory utilization. Starting with a baseline PP + ZeRO-2 implementation in TorchTitan, we incrementally add: (1) activation offloading (O), (2) interleaved pipelining and optimizer updates (I), and (3) heterogeneous pipeline parallelism (H). We evaluate these components using Llama 65B on Cluster A, with results shown in Figure 11.

The PP + ZeRO-2 baseline fails to train the model due to excessive activation memory overhead. Adding activation offloading (O) enables training with a PP degree of 5 and ZeRO-2 degree of 4, but H100 GPUs remain underutilized despite being 3× faster than A100s, as they lack sufficient memory to store proportionally more layers. Interleaved pipelining (I) further reduces memory utilization by 26% due to parameter offloading and interleaved optimizer updates. Finally, with heterogeneous pipeline parallelism (H), we reduce pipeline stages to 4 by grouping GPUs within each node. This configuration achieves balanced computation across GPUs, trading slightly higher memory utilization for significant performance improvements.

6.7 Planner Optimization Time

The search space for optimal training configurations in Zorse is extensive, encompassing asymmetric combinations of PP and DP, model partitioning, interleaved pipelining, and microbatch sizes. Despite this complexity, Zorse's planner completes within 3 minutes for all workloads. This is a negligible overhead to pay for optimizing LLM training, which typically requires hundreds of thousands of GPU hours [49]. Fast planning is achieved through a two-phase optimization approach that employs approximation algorithms and heuristics to effectively prune the search space. Figure 12 presents a breakdown of optimization time for the largest models in each cluster. While profiling dominates planning time, it scales sublinearly with the number of GPUs, since we make optimizations to parallelize profiling and avoid redundant measurements on duplicate VMs and GPUs.

7 Related Work

Heterogeneous Training. Several systems have been proposed to optimize training on heterogeneous clusters with data parallelism (DP) [15, 19, 27, 31], pipeline parallelism (PP) [4, 34, 57], tensor parallelism (TP) [59], and a combination of all three [12, 50, 55]. Existing strategies perform suboptimally when GPU memory and networking are limited. DP-only systems [9, 19, 27, 60] underutilize faster links as they are bottlenecked by slower links during collective communication. PP with DP reduces communication overhead but cannot efficiently utilize memory-efficient ZeRO-3 DP without significant communication costs. TP [50, 55, 59] is generally ineffective in heterogeneous environments due to high communication overhead (Section 2.1). Systems combining approaches inherit limitations of each individual approach. Systems like MiCS [53, 56, 61] optimize for heterogeneous networks, but lack support for clusters with compute and memory heterogeneity. Zorse integrates PP and DP with optimizations enabling both communication and memory-efficient training while balancing resource heterogeneity, delivering superior performance on heterogeneous clusters.

Pipelining Optimizations. [29] introduces interleaved pipelining, using 1F1B scheduling that interleaves forward and backward computations from different ministages. When combined with ZeRO [38], this approach either incurs additional communication for parameter fetching, or requires storing all ministage parameters in memory. [37] presents an optimized schedule that eliminates pipeline bubbles, but requires keeping all ministage parameters in memory, and without interleaving, optimizer updates cannot overlap with computation, incurring additional overhead. Zorse uses interleaved pipelining with a GPipe-style [11] schedule, integrating with ZeRO-2 for communication efficiency while achieving superior memory efficiency through parameter and activation offloading, and interleaved optimizer updates.

Memory Optimizations. Various works propose memory optimization strategies including activation checkpointing, activation offloading, and parameter offloading [9, 14, 35, 39, 42]. Mist [64] jointly optimizes these techniques with parallelism strategy, but does not address heterogeneous clusters. Zorse uniquely optimizes both checkpointing and offloading while targeting heterogeneous training environments.

8 Discussion and Future Work

Dynamic Parallelism. Hardware performance variability and node failures present challenges, especially in cloud environments [13, 23, 41]. Although Zorse determines parallelism configuration statically, its low-overhead planner enables periodic reconfiguration with updated performance models to accommodate hardware changes. Zorse could also incorporate existing approaches for efficient parallelism adaptation [7, 25, 52] and node failure recovery [2, 5, 6].

Tensor Parallelism. Although Zorse does not support TP due to its complexity and limited suitability for heterogeneous clusters, TP remains valuable for specific scenarios with very large models, extended context lengths, or small batch sizes. Future work could extend Zorse to incorporate

DP + TP (including sequence parallelism [21]) combinations within GPU groups.

Generalizability. Like previous studies [9, 19, 50, 55, 59], Zorse uses NVIDIA GPUs for evaluation due to their widespread availability and usage in training [24]. Nevertheless, Zorse's implementation is adaptable to any accelerator with PyTorch support, including AMD GPUs [1] and TPUs [10, 18]. Furthermore, its design principles are hardware-agnostic and applicable to other heterogeneous environments.

9 Conclusion

In this paper, we present Zorse, a system that allows users to train large language models efficiently on clusters with significant compute, memory, and networking heterogeneity. Zorse achieves this by optimizing the combination of PP with DP such that it is both memory efficient and incurs low communication overhead while balancing heterogeneous hardware utilization. Moreover, it uses a planner to efficiently find an optimized training strategy from the vast search space of possible training configurations for heterogeneous clusters. Our experiments demonstrate that Zorse can achieve up to 3× speedup in training throughput compared to existing state-of-the-art systems on three diverse and representative heterogeneous training scenarios.

References

[1] AMD. 2025. *PyTorch on ROCm.* https://rocm.docs.amd.com/projects/install-on-linux/en/latest/install/3rd-party/pytorch-install.html Accessed: 2025-02-24.

[2] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 472–487. https://doi.org/10.1145/3492321.3519584

[3] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sashank Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2023. PaLM: scaling language modeling with pathways. *J. Mach. Learn. Res.* 24, 1, Article 240 (Jan. 2023), 113 pages.

[4] Yushi Ding, Noam Botzer, and Tim Weninger. 2021. HetSeq: Distributed GPU Training on Heterogeneous Infrastructure. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. AAAI Press, 15432–15438. https://doi.org/10.1609/aaai.v35i17.17813

[5] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. 2024. Parcae: proactive, liveput-optimized DNN training on preemptible instances. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'24)*. USENIX Association, USA, Article 62, 19 pages.

[6] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. 2024. ReCycle: Resilient Training of Large DNNs using Pipeline Adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 211–228. https://doi.org/10.1145/3694715.3695960

[7] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, Xupeng Miao, and Bin Cui. 2024. Enabling Parallelism Hot Switching for Efficient Training of Large Language Models. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 178–194. https://doi.org/10.1145/3694715.3695969

[8] Oded Goldschmidt and Dorit S. Hochbaum. 1988. A Polynomial Algorithm for the k-Cut Problem for Fixed k. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 444–451.

[9] Runsheng Benson Guo, Utkarsh Anand, Arthur Chen, and Khuzaima Daudjee. 2024. Cephalo: Harnessing Heterogeneous GPU Clusters for Training Transformer Models. arXiv:2411.01075 [cs.DC] https://arxiv.org/abs/2411.01075

[10] Ronghang Hu, Vaibhav Singh, Jack Cao, Milad Mohammadi, Yeounoh Chung, Shauheen Zahirazami, and Ross Girshick. 2022. Scaling PyTorch Models on Cloud TPUs with FSDP. https://pytorch.org/blog/scaling-pytorch-models-on-cloud-tpus-with-fsdp/ Accessed: 2025-02-24.

[11] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[12] Sunyeol Hwang, Eungyeong Lee, Hongseok Oh, and Youngmin Yi. 2024. FASOP: Fast yet Accurate Automated Search for Optimal Parallelization of Transformers on Heterogeneous GPU Clusters. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing* (Pisa, Italy) *(HPDC '24)*. Association for Computing Machinery, New York, NY, USA, 253–266. https://doi.org/10.1145/3625549.3658687

[13] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. 2011. On the Performance Variability of Production Cloud Services. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 104–113. https://doi.org/10.1109/CCGrid.2011.22

[14] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems 2020*. 497–511.

[15] 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. 2022. Whale: Efficient Giant Model Training over Heterogeneous GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 673–688. https://www.usenix.org/conference/atc22/presentation/jia-xianyan

[16] Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Guoliang He, Xupeng Miao, Ana Klimovic, Bin Cui, Binhang Yuan, and Eiko Yoneki. 2025. Demystifying Cost-Efficiency in LLM Serving over Heterogeneous GPUs. arXiv:2502.00722 [cs.DC] https://arxiv.org/abs/2502.00722

[17] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li,

Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: scaling large language model training to more than 10,000 GPUs. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'24)*. USENIX Association, USA, Article 41, 16 pages.

[18] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 82, 14 pages. https://doi.org/10.1145/3579371.3589350

[19] Kyeonglok Kim, Hyeonsu Lee, Seungmin Oh, and Euiseong Seo. 2022. Scale-Train: A Scalable DNN Training Framework for a Heterogeneous GPU Cloud. *IEEE Access* 10 (2022), 68468–68481.

[20] Joel Lamy-Poirier. 2021. Layered gradient accumulation and modular pipeline parallelism: fast and efficient training of large language models. *arXiv preprint arXiv:2106.02679* (2021).

[21] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. 2023. Sequence Parallelism: Long Sequence Training from System Perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 2391–2404. https://doi.org/10.18653/v1/2023.acl-long.134

[22] 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

[23] Liang Luo, Peter West, Pratyush Patel, Arvind Krishnamurthy, and Luis Ceze. 2022. Srifty: Swift and Thrifty Distributed Neural Network Training on the Cloud. In *Proceedings of Machine Learning and Systems (MLSys)*, Vol. 4. 833–847. https://proceedings.mlsys.org/paper_files/paper/2022/file/0cafb7890f6a7d4de65507d5bb7e0187-Paper.pdf

[24] MarketsandMarkets. 2023. Nvidia's Dominance in the AI Chip Market. https://www.marketsandmarkets.com/blog/SE/nvidia-dominance-in-the-ai-chip-market

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

[26] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. 2022. Galvatron: Efficient Transformer Training over Multiple GPUs Using Automatic Parallelism. *Proc. VLDB Endow.* 16, 3 (nov 2022), 470–479. https://doi.org/10.14778/3570690.3570697

[27] Sergio Moreno-Alvarez, Juan M Haut, Mercedes E Paoletti, Juan A Rico-Gallego, Juan C Diaz-Martin, and Javier Plaza. 2020. Training deep neural networks: a static load balancing approach. *The Journal of Supercomputing* 76 (2020), 9739–9754.

[28] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[29] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*. PMLR, 7937–7947.

[30] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[31] Chengyi Nie, Jessica Maghakian, and Zhenhua Liu. 2024. Cannikin: Optimal Adaptive Distributed DNN Training over Heterogeneous Clusters. In *Proceedings of the 25th International Middleware Conference* (Hong Kong, Hong Kong) *(Middleware '24)*. Association for Computing Machinery, New York, NY, USA, 299–312. https://doi.org/10.1145/3652892.3700767

[32] NVIDIA. 2024. NCCL: NVIDIA Collective Communications Library. https://developer.nvidia.com/nccl.

[33] NVIDIA Corporation. 2020. Point To Point Communication Functions. NVIDIA NCCL User Guide API documentation, version 2.26.2. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/api/p2p.html Accessed: 2025-04-17.

[34] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. 2020. {HetPipe}: Enabling large {DNN} training on (whimpy) heterogeneous {GPU} clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 307–321.

[35] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 891–905. https://doi.org/10.1145/3373376.3378505

[36] PyTorch. 2023. Training a 1 Trillion Parameter Model with PyTorch Fully Sharded Data Parallel on AWS. https://shorturl.at/6Y4LT. Accessed: 2024-01-30.

[37] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. 2024. Zero bubble (almost) pipeline parallelism. In *The Twelfth International Conference on Learning Representations*.

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

[39] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. 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* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 59, 14 pages. https://doi.org/10.1145/3458817.3476205

[40] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[41] M. Suhail Rehman and Majd F. Sakr. 2010. Initial Findings for Provisioning Variation in Cloud Computing. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CLOUDCOM '10)*. IEEE Computer Society, USA, 473–479. https://doi.org/10.1109/CloudCom.2010.47

[42] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 551–564. https://www.usenix.org/conference/atc21/presentation/ren-jie

[43] H. Robbins and S. Monro. 1951. A stochastic approximation method. *The Annals of Mathematical Statistics* 22, 3 (1951), 400–407.

[44] Huzur Saran and Vijay V. Vazirani. 1995. Finding k Cuts within Twice the Optimal. *SIAM J. Comput.* 24, 1 (1995), 101–108. https://doi.org/10.1137/S0097539792251730 arXiv:https://doi.org/10.1137/S0097539792251730

[45] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. 2018. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems* 31 (2018).

[46] 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. *CoRR* abs/1909.08053 (2019). http://arxiv.org/abs/1909.08053

[47] Mechthild Stoer and Frank Wagner. 1997. A Simple Min-Cut Algorithm. *J. ACM* 44, 4 (1997), 585–591. https://doi.org/10.1145/263867.263872

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

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

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

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

[52] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. 2024. Tenplex: Dynamic Parallelism for Deep Learning using Parallelizable Tensor Collections. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 195–210. https://doi.org/10.1145/3694715.3695975

[53] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Xiaoxia Wu, Connor Holmes, Zhewei Yao, Samyam Rajbhandari, Olatunji Ruwase, Feng Yang, Lei Yang, and Yuxiong He. 2024. ZeRO++: Extremely Efficient Collective Communication for Large Model Training. In *ICLR 2024*. https://www.microsoft.com/en-us/research/publication/zero-extremely-efficient-collective-communication-for-large-model-training/

[54] Yifu Wang, Horace He, Less Wright, Luca Wehrstedt, Tianyu Liu, and Wanchao Liang. 2024. Distributed w/ TorchTitan: Introducing Async Tensor Parallelism in PyTorch. https://discuss.pytorch.org/t/distributed-w-torchtitan-introducing-async-tensor-parallelism-in-pytorch/209487 Accessed: 2025-02-24.

[55] Ran Yan, Youhe Jiang, Xiaonan Nie, Fangcheng Fu, Bin Cui, and Binhang Yuan. 2025. HexiScale: Accommodating Large Language Model Training over Heterogeneous Environment. arXiv:2409.01143 [cs.DC] https://arxiv.org/abs/2409.01143

[56] Fei Yang, Shuang Peng, Ning Sun, Fangyu Wang, Yuanyuan Wang, Fu Wu, Jiezhong Qiu, and Aimin Pan. 2024. Holmes: Towards Distributed Training Across Clusters with Heterogeneous NIC Environment. In *Proceedings of the 53rd International Conference on Parallel Processing* (Gotland, Sweden) *(ICPP '24)*. Association for Computing Machinery, New York, NY, USA, 514–523. https://doi.org/10.1145/3673038.3673095

[57] Jinghui Zhang, Geng Niu, Qiangsheng Dai, Haorui Li, Zhihua Wu, Fang Dong, and Zhiang Wu. 2023. PipePar: Enabling fast DNN pipeline parallel training in heterogeneous GPU clusters. *Neurocomput.* 555, C (Oct. 2023), 12 pages. https://doi.org/10.1016/j.neucom.2023.126661

[58] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. TinyLlama: An Open-Source Small Language Model. arXiv:2401.02385 [cs.CL]

[59] Shiwei Zhang, Lansong Diao, Chuan Wu, Zongyan Cao, Siyu Wang, and Wei Lin. 2024. HAP: SPMD DNN Training on Heterogeneous GPU Clusters with Automated Program Synthesis. In *Proceedings of the European Conference on Computer Systems (EuroSys '24)* (Athens, Greece, April 22–25). ACM, New York, NY, USA, 18. https://doi.org/10.1145/3627703.3629580

[60] WenZheng Zhang, Yang Hu, Jing Shi, and Xiaoying Bai. 2025. Poplar: Efficient Scaling of Distributed DNN Training on Heterogeneous GPU Clusters. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 22587–22595.

[61] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. 2022. MiCS: near-linear scaling for training gigantic model on public cloud. *Proc. VLDB Endow.* 16, 1 (Sept. 2022), 37–50. https://doi.org/10.14778/3561261.3561265

[62] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3848–3860. https://doi.org/10.14778/3611540.3611569

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

[64] Zhanda Zhu, Christina Giannoula, Muralidhar Andoorveedu, Qidong Su, Karttikeya Mangalam, Bojian Zheng, and Gennady Pekhimenko. 2025. Mist: Efficient Distributed Training of Large Language Models via Memory-Parallelism Co-Optimization. In *Proceedings of the Twentieth European Conference on Computer Systems*. 1298–1316.