

ZenFlow: Enabling Stall-Free Offloading for LLM Training

TINGFENG LAN, University of Virginia, USA

YUSEN WU, University of Virginia, USA

BIN MA, University of California, Merced, USA

ZHAOYUAN SU, University of Virginia, USA

RUI YANG, University of Virginia, USA

TEKIN BICER, Argonne National Laboratory, USA

MASAHIRO TANAKA, AnyScale, USA

OLATUNJI RUWASE, Snowflake, USA

DONG LI, University of California, Merced, USA

YUE CHENG, University of Virginia, USA

Fine-tuning large language models (LLMs) often exceeds GPU memory limits, prompting systems to offload model states to CPU memory. However, existing offloaded training frameworks like ZeRO-Offload treat all parameters equally and update the full model on the CPU, causing severe GPU stalls, where fast, expensive GPUs sit idle, waiting for slow CPU updates and limited-bandwidth PCIe transfers.

We present ZenFlow, a new offloaded training framework that decouples updates between GPU and CPU by prioritizing important gradients. ZenFlow performs in-place updates of important gradients on GPU while offloading and accumulating less important ones on CPU without blocking training iterations, fully overlapping CPU work with GPU computation. To make this importance-aware design scalable, ZenFlow leverages a novel spatio-temporal locality of important gradients—concentrated in specific positions that remain stable across iterations—to make important gradient identification efficient and lightweight while preserving fidelity. Evaluation shows that ZenFlow achieves up to 5× end-to-end speedup, 2× lower PCIe traffic, 85% less GPU stalls, and up to 72% lower cloud GPU cost, all while preserving accuracy.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Computer systems organization** → *Heterogeneous (hybrid) systems*; • **Software and its engineering** → *Software performance*.

Additional Key Words and Phrases: large language model training, CPU offloading, distributed training, heterogeneous computing

ACM Reference Format:

Tingfeng Lan, Yusen Wu, Bin Ma, Zhaoyuan Su, Rui Yang, Tekin Bicer, Masahiro Tanaka, Olatunji Ruwase, Dong Li, and Yue Cheng. 2026. ZenFlow: Enabling Stall-Free Offloading for LLM Training. *Proc. ACM Manag. Data* 4, 4 (SIGMOD), Article 292 (September 2026), 28 pages. <https://doi.org/10.1145/3837130>

Authors' Contact Information: Tingfeng Lan, erc8gx@virginia.edu, University of Virginia, Charlottesville, Virginia, USA; Yusen Wu, xrn4ub@virginia.edu, University of Virginia, Charlottesville, Virginia, USA; Bin Ma, bma100@ucmerced.edu, University of California, Merced, Merced, California, USA; Zhaoyuan Su, acf7ea@virginia.edu, University of Virginia, Charlottesville, Virginia, USA; Rui Yang, qgh4hm@virginia.edu, University of Virginia, Charlottesville, Virginia, USA; Tekin Bicer, tbicer@anl.gov, Argonne National Laboratory, Lemont, Illinois, USA; Masahiro Tanaka, tanaka.masahiro@gmail.com, AnyScale, San Francisco, California, USA; Olatunji Ruwase, tunji.ruwase@snowflake.com, Snowflake, Seattle, Washington, USA; Dong Li, dli35@ucmerced.edu, University of California, Merced, Merced, California, USA; Yue Cheng, mrz7dp@virginia.edu, University of Virginia, Charlottesville, Virginia, USA.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2836-6573/2026/9-ART292

<https://doi.org/10.1145/3837130>

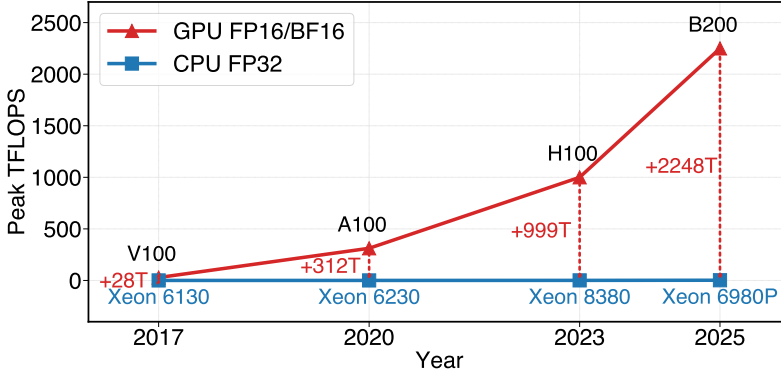


Fig. 1. GPU vs. CPU peak performance evolution.

1 Introduction

Large language models (LLMs) have become the foundation of modern natural language processing applications, powering tasks from text generation to code synthesis [1, 30, 48, 66]. While pretrained models [8, 48, 62, 63] offer general-purpose capabilities, fine-tuning them on domain-specific data is often essential for achieving high performance on downstream tasks. However, as LLM sizes grow, the memory demands of fine-tuning far exceed the capacity of a single GPU, making large-scale training increasingly challenging [5, 13, 56].

To address this memory bottleneck, offloading-based training systems, such as ZeRO-Offload [53], ZeRO-Infinity [50], and StrongHold [61], have emerged as promising solutions. These systems reduce GPU memory consumption by offloading model states (e.g., gradients and optimizer states) to CPU memory or NVMe storage and performing optimizer updates on the CPU. However, the CPU-GPU performance gap has widened rapidly (see Fig. 1), increasing from 28 TFLOPS to 2248 TFLOPS in the last couple of years, so CPU-side optimizer updates and PCIe transfers are largely exposed to the critical path, causing long GPU stalls (Fig. 2). For example, on fine-tuning a Llama-2-7B [63] model with $4 \times$ A100 GPUs, one training step time experiences a dramatic increase from 0.5s to 7s when enabling offloading, a $14 \times$ slowdown.

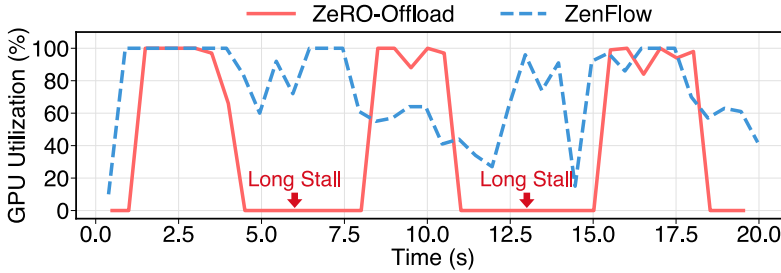


Fig. 2. Comparison of GPU utilization over time.

We identify two major sources of inefficiency in existing offloaded training systems: (1) *Long stalls caused by slow CPU-side updates*, which delay the next training iteration and leave GPUs idle. (2) *High I/O cost*, as each iteration requires transferring the full set of gradients and updated parameters between GPU and CPU.

These performance bottlenecks stem from critical limitations of current offloading systems: (1) They adopt a uniform strategy that treats all parameters and gradients as equally important, regardless of how much each individual gradient actually contributes to learning. (2) They rely on slow CPU with limited computation and PCIe bandwidth for updating and transferring entire model states.

In practice, this causes slow CPU to bottleneck fast and more expensive GPU. As shown in Fig. 2, during three consecutive training steps, ZeRO-Offload suffers from repeated GPU stalls—each lasting up to 5 seconds within a 7-second step—where GPU utilization drops to nearly 0%. These prolonged idle periods dominate the training time, resulting in severe underutilization of GPU resources.

However, our analysis and former study [2, 3, 27, 39] reveal that *LLM gradients are highly imbalanced*: the top 1% of gradients account for over 90% of the total gradient norm. Consequently, existing hardware-agnostic treatment of gradients leads to wasted computation and communication on low-impact updates, stalling overall training throughput.

We introduce ZenFlow, the first offloaded training system that decouples parameter updates based on both hardware heterogeneity and learning dynamics (e.g., the gradient importance). The key insight behind ZenFlow is to *treat important and unimportant updates differently*: by retaining the small but critical subset of gradients on GPU for immediate updates, ZenFlow avoids GPU stalls and ensures **fast** updates on important parameters. Meanwhile, remaining less important gradients are offloaded to the **slower** CPU, where they are slowly accumulated and updated. This strategy preserves the learning contribution of less important gradients while avoiding frequent update and I/O cost. By amortizing CPU-side updates over multiple GPU iterations, ZenFlow effectively combines the high speed of GPU with the cost efficiency of CPU. As illustrated in Fig. 2, this design enables ZenFlow to eliminate repeated GPU stalls seen in ZeRO-Offload, resulting in consistently high GPU utilization and significantly improved end-to-end efficiency.

A key challenge in realizing ZenFlow is determining which parameters are important enough to be updated immediately on GPUs. This becomes particularly difficult in distributed training settings such as ZeRO [49], where each GPU holds a shard of the full model and its corresponding gradients. Performing an AllGather to collect the full gradient matrix across GPUs would incur prohibitive communication and memory cost. To address this, ZenFlow leverages a novel observation: *to our best knowledge, we are the first to identify that important gradients in LLM fine-tuning exhibit strong spatial and temporal locality*. Specifically, we find that a small subset of input dimensions (i.e., channels) consistently carries high-magnitude gradients across training steps. By tracking and reusing this compact set of important channels, ZenFlow enables efficient, scalable importance-aware training without expensive global synchronization.

This paper makes the following contributions:

- We discover and characterize a novel spatio-temporal locality property of important gradients in LLM fine-tuning.
- We design a lightweight method to identify important gradients in distributed training without costly global synchronization.
- We build ZenFlow, a fine-grained CPU-GPU pipeline that decouples parameter updates with different importance to minimize GPU stalls and I/O overhead.
- We build ZenFlow on DeepSpeed [51] and evaluate it thoroughly across single-GPU and multi-GPU settings. ZenFlow achieves 3.6-5× end-to-end speedup compared to state-of-the-art offloading systems while maintaining accuracy across diverse LLMs and fine-tuning tasks.

Open-source release. ZenFlow has been integrated into and is open-sourced at [deepspeedai/DeepSpeed](https://github.com/deepspeedai/DeepSpeed).

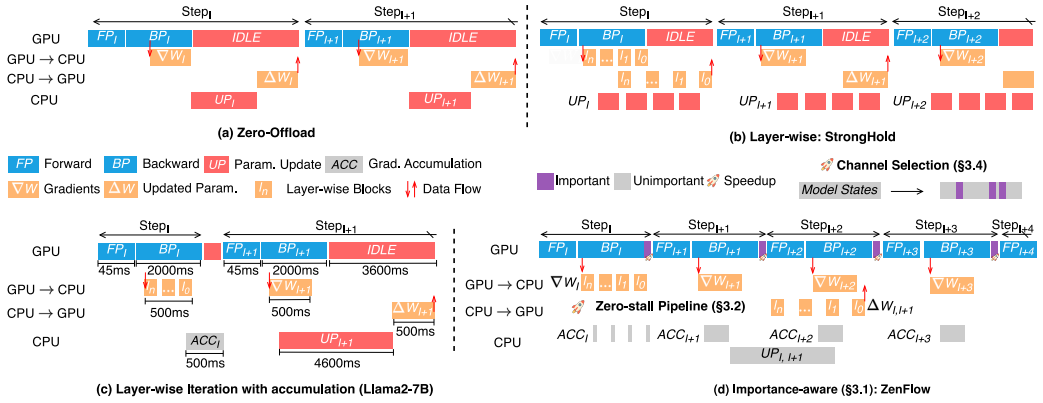


Fig. 3. Offloading strategy comparison. (a) **ZeRO-Offload** [53] sequentially executes FP and BP on GPU, then offloads gradients and performs UP on CPU, leaving the GPU idle. (b) **StrongHold** [61] overlaps CPU updates with GPU backward computation and gradient offloading by offloading and updating gradients layer by layer. However, CPU-side update is still too slow to hide, causing GPU stalls. (c) **Example: iteration with accumulation:** Update occurs every two steps—one for accumulation (i.e., gradients are summed without applying updates) (ACC), one for normal update (UP). (d) **ZenFlow** prioritizes parameter updates for important gradient (§3.3) on fast GPU to leverage its high compute bandwidth, while accumulating unimportant gradients in several rounds on slow CPU to reduce unnecessary parameter update overhead (§3.1). This design decouples fast GPU computing from slow CPU processing, minimizing GPU stalls and reducing data movement (§3.2).

2 Background and Motivation

2.1 Distributed Training Systems

Basics of Distributed Training. Deep learning model training typically consists of millions of iterations performed across multiple training epochs [8, 62, 63]. Each iteration mainly involves three stages: forward propagation (FP), backward propagation (BP), and parameter update (UP). In the FP stage, a batch of training data is passed through the model to compute the output and loss based on an objective function. In the BP stage, the model propagates the loss value reversely through model layers to compute gradients for each model parameter. Finally, in the UP stage, model parameters are updated using the computed gradients through an optimization algorithm by the optimizer (e.g., SGD [7], Adam [31]).

Distributed Training. To train large models efficiently, distributed parallelism is widely adopted. When models fit in GPU memory, data parallelism [33] is commonly used, replicating the model across devices and sharding/distributing input batches. For larger models that exceed memory capacity, model parallelism [58] and pipeline parallelism [26, 45] partition model layers across devices to utilize aggregate memory. Furthermore, ZeRO [51] extends data parallelism by sharding model parameters, gradients, and optimizer states across devices. ZeRO can be regarded as a hybrid of data and model parallelism: each device processes a full batch in data parallelism style while applying model-parallelism-like partitioning to memory-intensive states. This design enables training models far beyond single-device memory limits and has become foundational in modern systems such as DeepSpeed [51], Megatron-LM [58], and PyTorch FSDP [75].

Table 1. Comparison of fine-tuning approaches.

Method	Parameter Updates	Memory Cost	Training Speed
Full Fine-tuning	Full (100%)	Very High (4M)	Low
PEFT (e.g., LoRA)	Partial (< 1%)	Low (M)	High
ZeRO-Offload	Full (100%)	Low (M)	Low
ZenFlow (ours)	Full (100%)	Low (M)	High

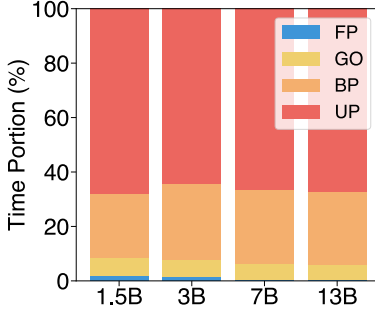


Fig. 4. Per-iteration time breakdown under ZeRO-Offload when training Qwen2.5-{1.5B,3B} and Llama2-{7B, 13B} models.

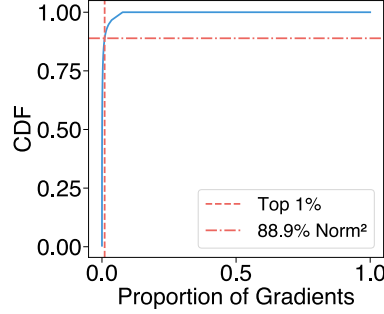


Fig. 5. CDF of gradient norm squared across all gradients on fine-tuning Qwen2.5-0.5B on the Alpaca52K dataset.

2.2 Memory-Efficient Training

Training memory is dominated by persistent states (i.e., parameters, gradients, and optimizer states), while activations are short-lived. In BF16/FP16, if parameters take M , gradients add another M , and optimizer states add $2M$, approximately $4M$ in total (see Table 2). For Llama2-7B, that is 14GB each for parameters and gradients, and 28GB for optimizer states—totaling 56GB, which easily exceeds the memory limit of an A100 40GB GPU.

Offloaded Training. To mitigate the GPU memory bottleneck, some offloading techniques [24, 50, 52, 53] have been proposed to transfer model parameters and optimizer states from the GPU memory to other storage, such as CPU DRAM or NVMe storage. Those methods reduce the GPU memory consumption and enable “out-of-core” training of larger models. ZeRO-Offload [53] in the DeepSpeed ZeRO series [49, 50, 53], is one of those state-of-the-art systems.

Fig. 3(a) illustrates the workflow of ZeRO-Offload. In each training iteration ($Step_I$), the forward pass (FP_I) and backward pass (BP_I) are executed on the GPU. During the backward pass, the gradients are computed and then transferred from GPU to CPU memory, where the CPU performs the parameter update (UP_I). Once the update is complete, the updated parameters ΔW_I are fetched back to the GPU for the forward pass in the next training iteration. During the UP stage (the parameter updating), the GPU is idle, waiting for the updated parameters to be fetched from the CPU.

Parameter-Efficient Fine-Tuning (PEFT). As summarized in Table 1, PEFT methods reduce training cost by updating only a small subset of parameters instead of all parameters. Typical PEFT approaches include adapters [21], prefix-tuning [34], LoRA [18, 23], and GaLore [74]. Among them, LoRA is the most widely used: it trains low-rank adapters while freezing the original weights, updating less than 1% of parameters. This design enables low memory usage and fast training.

Table 2. Resource requirement for fine-tuning Llama2-7B (BF16) on 4× A100 80GB GPUs with ZeRO-Offload.

Memory	Param. 14GB (<i>M</i>)	Optim. States 28GB (2 <i>M</i>)	Gradient 14GB (<i>M</i>)
Computation	FP on GPU 45ms/step	BP on GPU 2,000ms/step	UP on CPU 4,600ms/step
Communication	CPU-GPU Bandwidth ~28GB/s	Gradient Accumulation GPU→CPU 14GB (<i>M</i>)	Param. Update CPU→GPU 14GB (<i>M</i>)

However, this efficiency comes at the cost of representational fidelity. PEFT methods are structurally incapable of full-parameter adaptation, which can lead to suboptimal convergence and fail to capture the deep weight shifts required for domain specialization [10, 12, 15, 38, 47]. In this paper, we target full fine-tuning to maximize generality. Nevertheless, ZenFlow remains compatible with PEFT methods and can be combined when parameter efficiency is required.

2.3 Problems and Insights

Problem: Fast GPU execution is frequently stalled by slow CPU-side update. To understand the performance bottleneck of offloading-based fine-tuning, we break down per-iteration time when fine-tuning Qwen2.5 and Llama2 model series with various sizes (see Fig. 4). We use ZeRO-Offload with fully parallelized CPUAdam optimizer.

Each iteration is broken down into four stages: forward pass (FP), backward pass (BP), gradient offloading (GO), and parameter update (UP). With ZeRO-Offload, the forward and backward passes run on the GPU, the gradients are transferred to the CPU, and the update phase is computed entirely on the CPU with the CPUAdam optimizer. Despite parallelizing the CPUAdam optimizer across 128 CPU threads, the CPU-side update is a significant bottleneck (see Fig. 4). For example, when fine-tuning a Llama2-7B model, the update stage on the CPU takes ~4,600 *ms*—over twice as long as the backward pass time of 2,000 *ms*, which leads to GPU idling, as the GPU must wait for the CPU to complete its update computation before proceeding to the next iteration.

To improve GPU utilization, one natural solution from StrongHold [61] is to overlap CPU updates with GPU computation using layer-wise scheduling (Fig. 3 (b)). In this strategy, each layer’s gradients are offloaded and updated sequentially during the backward pass, allowing the CPU to begin updating earlier layers while the GPU continues computing later ones. This pipelined execution enables partial overlap between CPU-side updates and the GPU-side backward computation. For example, as soon as the gradient for layer l_n is computed, it can be offloaded and the corresponding update initiated on the CPU, rather than waiting for the full backward pass (for all remaining layers from l_{n-1} to l_0) to complete. This allows updated parameters to be uploaded back to the GPU earlier, reducing GPU idle time.

Although layer-wise scheduling can partially overlap CPU and GPU tasks, the CPU update phase is too long to be fully hidden (4,600 *ms* on the CPU vs. 2,000 *ms* on the GPU).

As a result, *GPU execution is frequently stalled, waiting for the CPU update even with aggressive multi-threaded optimization and optimal overlap.*

Furthermore, parameter upload introduces additional stalls. While gradient offloading overlaps naturally with the backward pass (since gradients can be streamed asynchronously to CPU), the next iteration must wait for updated parameters to return from CPU via PCIe. This adds additional 500 *ms* stall per iteration as shown in Fig. 3(c).

Next, we present insights to address the core limitation of existing offloaded training systems.

Insight #1: Gradients/parameters with different importance should be treated differently to decouple GPU/ CPU execution and reduce communication. It has been observed that the gradients of different parameters have different importance in deep neural network training [3, 19, 39, 60]. For example, Fig. 5 shows that top 1% of gradients account for ~90% of the gradient norm. However, the current offloading techniques *overlook* this difference in gradients, treating them equally and offloading all of them to the CPU regardless of their impacts. This uniform treatment introduces unnecessary inefficiencies. Take Llama2-7B as an example. An important subset of its parameters could be updated and transferred in ~50 ms, but under the uniform scheme they stall until the full CPU update (4,600 ms) and parameter transfer (500 ms) complete. In other words, the few important parameters that drive learning are forced to wait through ~5 s of unnecessary work dominated by unimportant ones.

This observation leads to a key research question: *Can we decouple important and less important gradients, and assign them to different hardware resources accordingly?* To mitigate unnecessary stalls, we propose updating important gradients directly on the GPU—leveraging its high computing bandwidth—while offloading and accumulating the remaining less important gradients on the CPU. This design relaxes the tight coupling between GPU execution and the full CPU update cycle, allowing the GPU to proceed without waiting on low-priority updates. The CPU-side accumulation proceeds *concurrently* and is typically fast enough (e.g., ~500 ms) to be hidden within the backward pass, thereby avoiding additional delay and improving GPU utilization.

Challenge: Full gradient view is expensive in fully sharded distributed training. To leverage gradient importance during training, selecting important gradients correctly and efficiently is a critical step.

A simple yet effective approach is top- k selection, which retains the gradients with the highest magnitudes. This strategy has been widely studied and applied in prior work [3, 20, 25, 27]. However, the top- k selection assumes access to the full set of gradients—what we refer to as a *global gradient view*.

In fully sharded distributed training, this assumption breaks. Each GPU holds only a fraction of the model parameters and computes gradients for its local shard, making global top- k selection *expensive*, because constructing a fully global view would require gathering and synchronizing gradients across all devices, incurring significant communication overhead and causing peak memory usage spikes due to gathered full gradient matrix. Therefore, directly applying existing global top- k selection is not practical.

Insight #2: Important LLM gradients show both spatial and temporal locality. To address the challenge posed by the lack of a global gradient view in fully sharded distributed training, we next investigate an important research question: *Do important gradients in LLM fine-tuning exhibit spatial or temporal locality that we can exploit to approximate global top- k selection more efficiently?*

To answer this, we analyze the gradient importance distribution during fine-tuning. As a representative case, we use Qwen2.5-0.5B on the Alpaca52K dataset to illustrate. We also evaluate whether this locality generalizes across model architectures and optimizers in §6.6.

First, we observe clear **spatial locality** in the gradient distribution. As shown in Fig. 6(a), the top 1% of gradients (highlighted in orange) are not uniformly distributed but instead concentrated in a narrow subset of columns, each corresponding to a specific input channel. This pattern persists across iterations (see Fig. 6(b) and discussed next).

This result suggests that a small set of input features consistently receive large updates during fine-tuning. This is because transformer activations are often localized across channels during inference [28, 57], and such patterns propagate backward to the gradients.

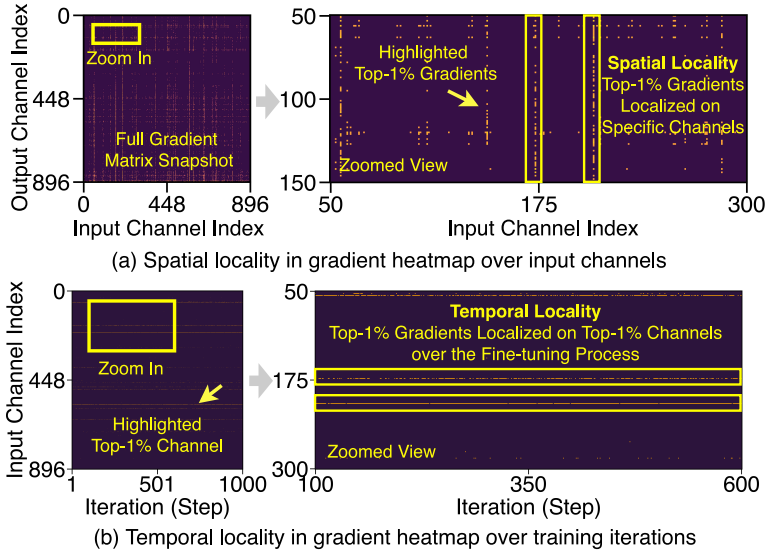


Fig. 6. Localities of gradients during fine-tuning.

Based on this, instead of performing the expensive global top- k selection over the entire parameter matrix, we can approximate gradient importance by identifying and tracking a small set of important input channels—an approach significantly cheaper in both computation and communication.

Second, we find that these important input channels also exhibit **temporal locality**. As shown in Fig. 6(b), the top 1% of important input-channel indices remain stable across iterations, forming persistent horizontal bands over time. This indicates that the same small subset of input channels consistently receives large updates during fine-tuning: these channels contribute more to the learning process than others, and their importance remains stable over time. While occasional deviations occur—likely due to exploration of alternative subspaces—the overall importance remains stable. Empirically, the top-1% channels exhibit a retention rate above 90%, while the retention rate approaches 100% under a looser threshold (e.g., top-5%). Therefore, we can efficiently track important gradients by maintaining a stable set of channels.

This spatio-temporal locality enables a lightweight approximation to global top- k selection: rather than recomputing gradient importance every iteration, we can approximate important gradients efficiently using a slowly-updated set of important channels. This drastically reduces communication and synchronization overhead, while preserving high fidelity in identifying important updates.

3 ZenFlow Design

The challenges and insights discussed in §2 motivate ZenFlow, a system that leverages gradient importance and gradients' spatio-temporal locality to mitigate the I/O bottleneck and GPU stalls in offloading training. ZenFlow decouples gradient updates across heterogeneous hardware and minimizes communication overhead without sacrificing accuracy. This section presents the design of ZenFlow with the following goals:

- **Goal #1.** Minimize GPU stalls caused by slow CPU (i.e., I/O and computation overhead associated with unimportant gradients and parameters).
- **Goal #2.** Ensure training reproducibility with deterministically scheduled update process.

- **Goal #3.** Guarantee training accuracy by preserving all (important and less important) gradient information.

3.1 Importance-aware Offloading Workflow

We begin by presenting the overall workflow of ZenFlow to provide a high-level view of how it decouples gradient updates across GPU and CPU. In this subsection, we assume that an important subset of gradients and their corresponding parameters have already been identified. The mechanism for efficiently selecting this subset is described later in §3.3. Here, we focus on how ZenFlow manages the importance-aware offloading and update process once the selection is in place.

Important updates: fast GPU path. As shown in Fig. 3(d), ZenFlow assigns the important gradient and parameters (highlighted in purple) to the GPU, while offloading the unimportant ones (shown in gray) to the CPU. At each iteration, ZenFlow performs a standard forward and backward pass with the full set of parameters. Once gradients are computed, the pre-identified important gradients remain on the GPU, where a *selective-optimizer*, initialized only with corresponding parameter subsets, performs an *in-place* update. This GPU-side update is lightweight, as it operates on a small subset of parameters, and it completes on the GPU without introducing stalls between iterations.

Unimportant updates: slow CPU path. Less important gradients are offloaded to the CPU and gradually accumulated over several iterations. These gradients are *not* discarded. They are simply delayed until they become important enough—a process of what we call *gradient accumulation*. Once the accumulated gradients become large enough to matter, ZenFlow performs a full parameter update on the CPU. In practice, this trigger is implemented via a fixed accumulation interval or an adaptive threshold (§3.2, §6.11), ensuring bounded staleness. This CPU-side update, though being slow, is carefully scheduled, so that it (1) perfectly overlaps with GPU computation to avoid stalls (see §3.2), and (2) ensures training reproducibility and accuracy (see §4).

In effect, ZenFlow updates important parameters more frequently using *fast* GPU, while updating less important ones less often on *slow* CPU, but with full gradient information.

This design achieves performance gains from three parts: (1) In each iteration, only a small subset of parameters are updated efficiently in place on GPU, and these updates are independent from the slow CPU-side. (2) The slow and compute-intensive CPU updates for large unimportant gradients are fully overlapped with multiple iterations of GPU computation, effectively hiding their latency. (3) Update I/O overhead for unimportant parameters is amortized as ZenFlow only updates and transfers them to GPU when they become large enough. Together, these optimizations minimize stalls.

3.2 Zero-stall Pipeline

While the importance-aware offloading design has the potential to minimize GPU stalls by selectively updating a small set of parameters, the CPU-side update can still become a bottleneck if not carefully overlapped with GPU computation. Moreover, naïvely creating an extra *selective-optimizer* on the GPU may incur unnecessary memory overhead. We next describe how ZenFlow addresses both challenges, by hiding CPU update latency and managing memory efficiently, to realize a truly zero-stall pipeline.

Modeling I/O Efficiency. We now provide an analytical comparison of the offloading workflow I/O traffic between ZenFlow and DeepSpeed ZeRO-Offload, demonstrating that ZenFlow’s pipeline significantly reduces communication overhead. For this analysis, we consider the case of fine-tuning with half-precision (BF16/FP16). In each iteration, ZeRO-Offload transfers the gradient generated from the GPU to the CPU, equivalent to one model copy (M). After the CPU optimizer completes the update, the updated parameters are transferred back to the GPU—another model copy (M). Therefore, the total I/O traffic per iteration is $2M$.

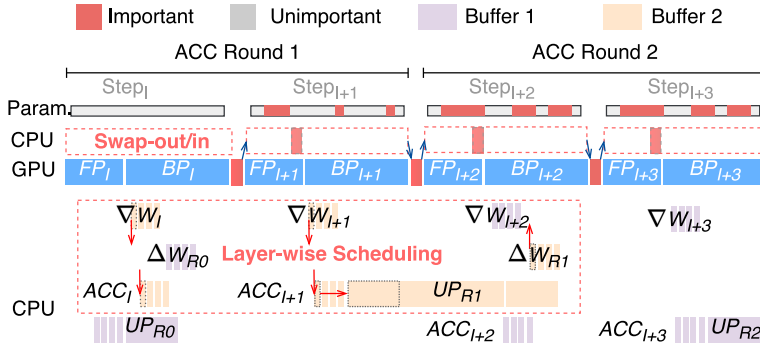


Fig. 7. Zero-stall pipelining with double buffering. ZenFlow overlaps GPU computation with CPU-side gradient accumulation and updates. Across training steps, important gradients are updated immediately on the GPU, while unimportant gradients are offloaded and accumulated on the CPU. The CPU maintains two buffers and alternates between accumulation (ACC) and update (UP) phases across rounds. While one buffer accumulates gradients from the current iterations, the other buffer applies parameter updates in the background. Layer-wise scheduling and swap-out/in ensure that CPU updates and parameter transfers are fully overlapped with GPU FP/BP, preventing GPU stalls.

In contrast, ZenFlow offloads only the gradients for less important parameters. Let k denote the top- k ratio (i.e., the fraction of gradients considered important) and S the number of accumulation rounds for the unimportant gradients. In each iteration, ZenFlow transfers only the $(1 - k) \cdot M$ unimportant gradients to the CPU. After S iterations, the CPU performs a parameter update and sends back the corresponding $(1 - k) \cdot M$ updated parameters. Therefore, the average I/O traffic per iteration in ZenFlow is: $\frac{(S+1) \cdot (1-k) \cdot M}{S}$.

Take $S = 4$ (one representative configuration of the accumulation rounds from our analysis for hiding CPU-side updates we will discuss shortly), and $k = 0.1$ as an example, the average I/O traffic per iteration becomes $1.125M$, which is nearly a $2\times$ reduction compared to ZeRO-Offload.

Hiding CPU-side Updates. In addition to reducing I/O traffic, ZenFlow also hides CPU-side update latency by overlapping it with the GPU computation. Our profiling shows that, for a Llama2-7B model, CPU update latency is approximately 4,600 ms with 128 CPU threads (fully parallelized on our GPU node) and $\sim 6,200$ ms with limited resources (e.g., with only 32 CPU threads). Uploading updated parameters to the GPU typically takes ~ 500 ms. In contrast, the GPU forward pass (~ 45 ms) and backward pass ($\sim 2,000$ ms) take around $\sim 2,045$ ms. This means the CPU update can be overlapped with $2.3\times$ to $3.1\times$ the GPU compute time.

Critically, we observe that unimportant gradients usually require 4-6 accumulation steps before crossing the importance threshold for update (see §6.11). This enables their CPU-side updates to be effectively masked by 4-6 iterations of GPU forward/backward computation, achieving near-complete overlap without additional delay. Empirically, we set the accumulation interval to 4 steps ($S = 4$), which is sufficient to hide CPU update latency in most cases while bounding staleness (see §4 for theoretical analysis).

To achieve this, ZenFlow employs a double-buffering strategy on the CPU. It maintains two gradient buffers: one for concurrently accumulating gradients from the current iteration, and one for holding previously accumulated gradients used for parameter updates. While the GPU executes the forward and backward passes and sends new gradients, the CPU concurrently applies updates accumulated in one buffer. See the step of UP_{R1} in Fig. 7, where parameters are updated using gradients accumulated from both $Step_I$ and $Step_{I+1}$.

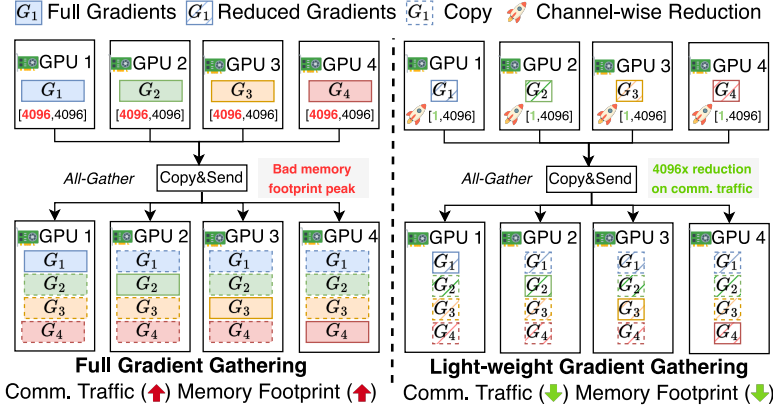


Fig. 8. Gradient gathering strategies. **Left:** Full gradient gathering collects and all-gathers full gradient matrices across GPUs, incurring high communication volume and large peak memory footprint. **Right:** ZenFlow performs light-weight gradient gathering by first reducing gradients along the channel dimension and exchanging only compact per-channel summaries.

Once the update is complete, the buffers are swapped, and the buffer used for parameter updates is cleared in order to store gradients in the next accumulation cycle. This design allows CPU updates to be completely hidden and run transparently in the background, fully overlapping with GPU training without introducing stalls.

Swapping out/in and Layer-wise Scheduling. To avoid excessive GPU memory consumption from the GPU-side *selective-optimizer*, ZenFlow swaps out its optimizer states to CPU and swaps back in before next update on GPU. These states are relatively small and can be transferred efficiently. To further reduce memory cost, swapping is performed in a layer-wise manner, ensuring that only one layer’s optimizer state resides on GPU at any time. This design maintains high memory efficiency by preventing additional overhead beyond what is required for a single layer. The same layer-wise scheduling is applied to gradient offloading and CPU-side updates to fully overlap communication and computation.

Hyperparameter Auto-tuning. To fully leverage the locality of important gradients while maintaining accuracy, we design ZenFlow-auto, which adaptively tunes the update interval based on observed learning dynamics. Specifically, ZenFlow-auto monitors gradient changes across GPUs using a lightweight coordination proxy (detailed in §3.3 and §6.11). For unimportant gradient part, ZenFlow-auto tracks the average accumulated channel gradient norm and compares it to the average one of the important part. Once the unimportant gradient part becomes comparable to important ones, ZenFlow-auto immediately triggers its CPU-side update, ensuring timely parameter refresh and stable convergence.

3.3 Gradient Selection

We now turn to the key question: *how to select important gradients for GPU-side updates in a scalable, low-overhead manner*. A natural approach is to prioritize gradients with large magnitudes, as they typically contribute more to learning. However, in fully sharded training, computing and comparing all gradient norms globally is expensive. To make this feasible in distributed training, ZenFlow introduces a lightweight yet effective approximation that leverages the spatio-temporal locality of important gradients discussed in §2.3.

Lightweight Proxy for Gradient Ranking. In fully sharded training, selecting important gradients across GPUs poses a major communication challenge. A straightforward solution is to rank all gradients globally by magnitude and prioritize the top- k for updates. However, this requires collecting all gradients across devices via expensive AllGather.

For example, consider fine-tuning Llama2-7B across 4 GPUs shown in Fig. 8(left). A naïve global ranking would require exchanging tens of gigabytes of gradients per iteration, incurring significant communication overhead. Even a single weight matrix imposes non-trivial communication overhead. Consider `q_proj.weight` in a transformer layer, shaped $[4096, 4096]$, containing 16.8M parameters. In a 4-way sharded setup, each device holds a $[1024, 4096]$ partition with 4.2M parameters. Using BF16/FP16, this amounts to 8MB per GPU. Aggregating gradients for this matrix across all devices requires 96MB of data transfer per iteration. Extrapolating to a 7B-parameter model, this results in a total gradient communication volume of 40GB per iteration.

To address this bottleneck, ZenFlow employs a communication efficient proxy: instead of gathering full $(n \times m)$ gradients, where n is the output dimension and m is the input dimension of a weight matrix, each GPU computes and shares per-column gradient norms squared (i.e., the sum of squared gradient values within each column). This reduces both communication and top- k selection complexity from $O(nm)$ to $O(m)$ while preserving gradient magnitude information and omitting only directional components. For example, consider a `q_proj.weight` from Llama2-7B with shape 4096×4096 . Rather than transferring the full 32MB gradient matrix in BF16, each GPU shares a 4096-dimensional vector (8KB), reducing communication volume by over 4,000 $\times$ with negligible impact on importance estimation (Fig. 8(right)).

Spatial Locality. We empirically validate the effectiveness of the approximation. Fig. 9(a) shows the cumulative distribution of top- k gradients and per-channel gradient norms during Qwen2.5-0.5B fine-tuning on Alpaca52K. Channels are sorted by how frequently they contain top- k gradients. The results show that 60-90% of top- k gradients are concentrated in just the top 10% of channels, and that per-channel gradient norms are strongly correlated with this top- k density. This confirms that channel-level summaries are a reliable indicator of gradient importance.

Temporal Locality. To further reduce the selection overhead, we examine whether important channels persist across iterations. Fig. 9(b) reports the retention ratio—the fraction of steps where previously selected channels remain within the important set. We find that once the top-1% of channels are tracked, the retention ratio already approaches 100%, and stays nearly perfect as the threshold widens to 3-5%. This strong temporal locality shows that caching and reusing channel indices is sufficient to cover important updates, avoiding repeated computation. In multi-GPU settings, this further reduces cross-device coordination and amortizes selection cost, enabling scalable importance-aware training.

4 Accuracy and Reproducibility Guarantee

To ensure that ZenFlow training delivers robust quality, we explicitly address the requirements of convergence and reproducibility across runs.

4.1 Convergence.

We now show that ZenFlow’s asynchronous offloading design does not affect the convergence property of existing optimizers.

Under standard assumptions commonly used in asynchronous optimization analysis [16, 35, 73, 76], we show that ZenFlow preserves the convergence rate of $O(1/\sqrt{T})$ with a bounded staleness factor where T is the total number of iterations. Such a convergence rate is the same as the ideal rate of synchronous SGD [16, 73].

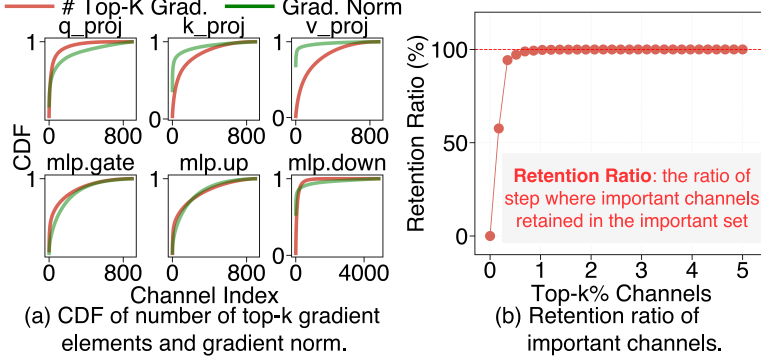


Fig. 9. CDF of top-k elements and gradient norm. The channel index is sorted by the number of top-k elements.

Why prioritizing high-norm channels. Prioritizing high-norm gradient channels is consistent with standard descent analysis. For an L -smooth objective, one SGD step satisfies

$$f(w_{t+1}) \leq f(w_t) - \eta \langle \nabla f(w_t), g_t \rangle + \frac{L\eta^2}{2} \|g_t\|_2^2.$$

Channels with larger gradient norm therefore have disproportionate influence on the descent direction. Existing top-K sparsification results further show that retaining large-norm channels with error compensation can preserve convergence behavior [4, 59].

Partial staleness in asynchronous training. We consider a mixed asynchronous training setup where the parameters are partitioned into two disjoint sets: $\theta = [\theta^{(g)}, \theta^{(c)}]$. The gradients w.r.t. $\theta^{(g)}$ are computed and applied *immediately* on the GPU every iteration, while the gradients w.r.t. $\theta^{(c)}$ are *accumulated on the CPU* over S iterations (typically $S = 4$) and then applied synchronously. Formally, at each training step t , we have the following.

$$\begin{aligned} \theta_{t+1}^{(g)} &= \theta_t^{(g)} - \alpha_t \nabla_{\theta^{(g)}} L(\theta_t) \\ \theta_{t+1}^{(c)} &= \begin{cases} \theta_t^{(c)} - \alpha_t \cdot \frac{1}{S} \sum_{i=t-S+1}^t \nabla_{\theta^{(c)}} L(\theta_i), & \text{if } t \bmod S = 0 \\ \theta_t^{(c)}, & \text{otherwise} \end{cases} \end{aligned}$$

where α_t is the learning rate and $L(\theta)$ is the objective loss function. The GPU handles the forward and backward passes for both $\theta^{(g)}$ and $\theta^{(c)}$, but only updates $\theta^{(g)}$ immediately. The CPU accumulates the gradients for $\theta^{(c)}$ over S iterations before applying the update.

Next, we model a partially stale update system with a bounded delay. Let $\rho = \frac{\sup_t \mathbb{E}[\|\nabla_{\theta^{(c)}} L(\theta_t)\|_2^2]}{\sup_t \mathbb{E}[\|\nabla L(\theta_t)\|_2^2]}$, representing the fraction of total gradient-norm energy that resides in the delayed coordinates (on the CPU). Empirically, we observe $\rho \approx 0.10$, as the GPU handles 90% of the gradient energy.

Bounded-staleness result. With common assumptions of unbiased stochastic gradients, bounded variance, and Lipschitz smoothness [16, 32, 35, 76], and letting ρ denote the fraction of gradient energy at the CPU side, we have the following:

$$\frac{1}{T} \sum_{t=1}^T \mathbb{E}[\|\nabla L(\theta_t)\|_2^2] \leq \mathcal{O}\left(\sqrt{\frac{1+\rho S}{T}}\right).$$

The term $O\left(\sqrt{\frac{1}{T}}\right)$ is the standard SGD rate; the factor $\sqrt{1 + \rho S}$ quantifies the extra cost of staleness. With $S = 4$ and $\rho \approx 0.10$, this factor is $\sqrt{1.4} \approx 1.18$, i.e., an 18 % slowdown relative to ideal synchronous SGD.

Warmup mitigates early-stage staleness. The convergence bound under the partial staleness includes a penalty term with the form $\sqrt{1 + \rho S}$, where ρ is the fraction of gradient-norm energy in the delayed coordinates and S is the accumulation interval. This bound assumes uniform gradient energy across training steps. However, in practice, the gradients are *not* uniformly distributed. Early training steps contribute disproportionately to optimization progress, as the gradient norms are significantly larger during this phase. Empirical and theoretical studies suggest that the gradient energy often decays as $\mathbb{E}[\|\nabla L(\theta_t)\|^2] \sim t^{-\beta}$, with $(0 < \beta < 1)$ [22, 29, 42].

ZenFlow exploits this observation by applying synchronous updates (i.e., no staleness) during the initial τ warmup steps, and then switches to asynchronous offloading for the remaining $T - \tau$ steps. This strategy eliminates staleness where it causes the most harm, while preserving efficiency later when gradients are smaller and more stable.

To quantify the effect, we compute the *gradient-weighted* penalty using a continuous approximation of the p -series.

$$\text{Penalty}(\beta) \approx \sqrt{1 + \rho S \cdot \left(1 - \left(\frac{\tau}{T}\right)^{1-\beta}\right)}.$$

This closed form captures the diminishing impact of the delayed gradients as training progresses.

For example, with a 5% warmup ($\tau/T = 0.05$), $S = 4$, $\rho = 0.1$, and $\beta = 0.6$ (typically ranging from 0.4 to 0.6 [29, 42]), the multiplicative penalty factor decreases from approximately 1.18 without warmup to 1.13 with warmup—equivalently, the excess penalty decreases from 18.3% to 13.1%.

Thus, the bound for ZenFlow incurs only about 13% overhead relative to the ideal SGD rate. Modern optimizers such as Adam and AdamW may further mitigate gradient staleness through momentum and adaptive learning rates [31, 39, 40, 55]. With this modest theoretical overhead, ZenFlow achieves up to 5 $\times$ end-to-end speedup, a 2 $\times$ reduction in I/O traffic, and near-zero stalls. We present the end-to-end speedup and a detailed breakdown of the performance gains in §6.

We use SGD in the derivation because it provides a general and transparent explanation of the delayed-update effect. While Adam and AdamW introduce momentum and adaptive per-coordinate scaling, ZenFlow relies on channel-level gradient-energy concentration rather than optimizer-specific update dynamics. Our cross-optimizer measurements in Table 4 further show that this locality remains strong under AdamW.

4.2 Determinism.

ZenFlow’s decoupled updating process is fundamentally different from traditional nondeterministic asynchronous training methods [33, 37, 68, 72], where update timing depends on variable factors such as system latency. ZenFlow ensures determinism: Given fixed seeds and configurations, the gradient selection, update and offloading schedules follow a *predetermined order*, making runs fully reproducible.

In multi-GPU training, ZenFlow overlaps important and unimportant channel computation only within each GPU; inter-GPU synchronization still follows standard ZeRO semantics with a barrier at every step. Consequently, per-stage execution latency does not affect the deterministic ordering of parameter updates observed across workers. We use deterministic CUDA kernels and apply a fixed channel-ranking pipeline. Together, these guarantees ensure that ZenFlow produces bit-exact identical training results under the same configuration.

5 Implementation

We build ZenFlow on DeepSpeed [51] with ~11K lines of Python code, without requiring any change to user training code. Our design extends DeepSpeed’s ZeRO-Offload [53] and ZeRO-Infinity [50] backends by integrating importance-aware gradient selection and *selective-optimizer* into the DeepSpeed runtime. We describe the key implementation components of ZenFlow below.

Fully Segmented Gradient Selection. In fully sharded distributed training, model states such as gradients are partitioned across GPUs. A gradient tensor may be flattened and split across multiple GPUs, so a single input-channel tensor can span two devices. To support fine-grained importance tracking, we introduce a segment mapping table that uses (segment_id, offset) tuples to index and manage gradient metadata. Each segment_id corresponds to one selected important channel. This structure enables flexible tracking of important channels during training even when they are scattered across devices. Additionally, we customize PyTorch’s flatten operations [46] to reorganize gradient storage from a row-major to a column-major layout, improving channel-wise access and manipulation.

Concurrent CPU-side Optimizer. To overlap GPU computation with CPU-side updates, ZenFlow initializes multiple CPU optimizer instances at setup. Each optimizer updates its assigned gradients—grouped into I/O-efficient buckets—as soon as data arrives from the GPU. Updates run in dedicated processes, with double-buffering and shared_memory enabling zero-copy communication. Concurrency is carefully managed to minimize synchronization overhead.

Selective GPU-side Optimizer. We extend PyTorch’s Adam and AdamW optimizers to support in-place updates using selected important gradients and enable fast swap-out and swap-in for extra optimizer state tensors.

6 Evaluation

6.1 Experimental Setup

Testbed. The overall experimental environment is summarized in Table 3. Our experiments are conducted on two server configurations: an A100 testbed and an H100 testbed. The A100 testbed consists of 4× NVIDIA A100 GPUs (80GB each), fully interconnected via NVLink, and paired with 64 CPUs and 1TB of CPU memory. The H100 testbed uses 4× NVIDIA H100 GPUs (80GB each) with NVLink, and paired with 64 CPUs and 2TB of CPU memory. Unless otherwise stated, experiments are conducted on the A100 testbed.

Models and Workloads. We evaluate ZenFlow across a diverse set of LLMs: OPT-350M, Qwen2.5-{0.5B, 1.5B, 3B}, and Llama2-{7B, 13B}, spanning from 350M to 13B parameters. All models follow their original architecture and default hyperparameters. We fine-tune these models on the widely adopted GLUE benchmark [64] following the standard setup in prior work [27], and expand the evaluation to larger and more diverse models to reflect real-world usage. The selected models represent some of the most popular choices in the open-source community [67], ensuring practical relevance. Unless otherwise noted, we train each model for 3 epochs with a batch size of 8 and a learning rate of $1e-5$. We use the AdamW optimizer [41] with a weight decay of 0.00 and apply a cosine learning rate schedule with 5% warmup. These settings are aligned with prior studies [11, 27, 74] to ensure fair, consistent comparison.

Full-parameter fine-tuning baselines. We compare ZenFlow (ZF) against SOTA offloading baselines: **ZeRO-Offload** (ZO) [53], **ZeRO-Infinity** [50], and **StrongHold** (SH) [61]. We configure ZeRO-Offload with ZeRO Stage 2 [49] to maximize training throughput. For ZeRO-Infinity, we use default ZeRO Stage 3 [49, 50] and disable NVMe offloading to avoid potential performance degradation caused by frequent SSD access during training. We implement StrongHold’s layer-wise scheduling technique on top of DeepSpeed ZeRO-Offload because it is not open-sourced.

Table 3. Experimental environments.

		A100 Testbed	H100 Testbed
HW	GPU	NVIDIA A100 (80GB) × 4	NVIDIA H100 (80GB) × 4
	CPU	AMD EPYC 7742	Intel Xeon Platinum 8462Y
	Memory	64C 128T (SMT Enabled)	64C 128T (SMT Enabled)
	PCIe	32×32GB DDR4-3200	32×64GB DDR5-4800
SW	Python / PyTorch	3.10 / 2.5.1	
	CUDA / DeepSpeed	11.8.0 / 0.16.2	
	Model	OPT-350M, Qwen2.5-{0.5B, 1.5B, 3B},	
		Llama-2-{7B, 13B}	

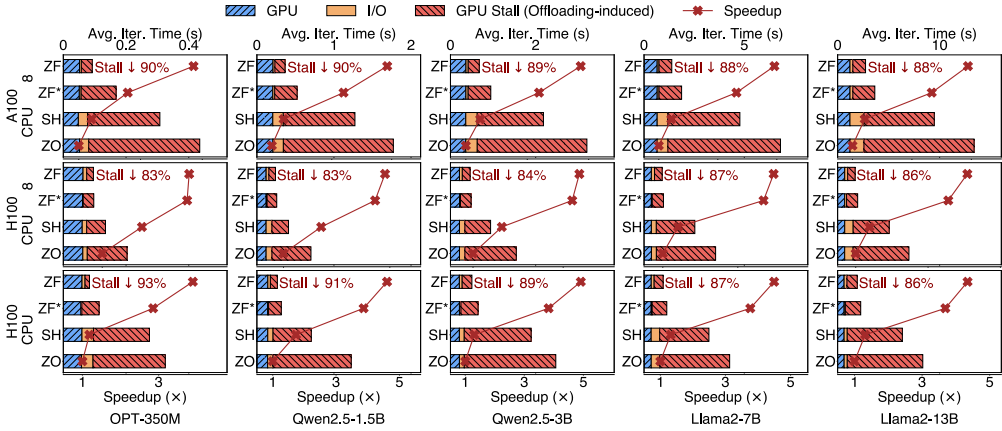


Fig. 10. Training time breakdown and speedup across hardware configurations and models. Bars correspond to the upper X-axis. Lines (with markers) show relative speedup over ZeRo-Offload (ZO), referenced on the bottom X-axis. SH denotes StrongHold, ZF* denotes ZenFlow* (importance-aware updates without zero-stall pipelining), and ZF denotes ZenFlow (full system).

PEFT baseline. We include GaLore [74] as a memory-saving PEFT baseline, using rank 128, batch size 8, and learning rate $4e-5$ with the default scale 0.25 to match ZenFlow’s effective $1e-5$ update scale. Notably, ZenFlow is orthogonal to these approaches and can be integrated with existing offloading strategies to further enhance scalability and performance—for example, by combining with gradient compression techniques or enabling deeper offloading with computational storage devices [27].

ZenFlow Variants. We evaluate two variants of ZenFlow to isolate the contributions of its key design components. **ZenFlow** represents the complete design of ZenFlow. It integrates two major optimizations: (1) *importance-aware offloading*, which selectively delays updates of unimportant parameters to reduce GPU stalls caused by CPU updates; and (2) *zero-stall pipeline*, which overlaps CPU-side optimizer updates with GPU computation and enables fast swap-out/in of *selective-optimizer* states on the GPU to avoid memory footprint peaks (§3.2). **ZenFlow*** is a simplified variant that *disables* the zero-stall pipeline while retaining importance-aware selective updates. This configuration isolates the performance benefit of pipelining.

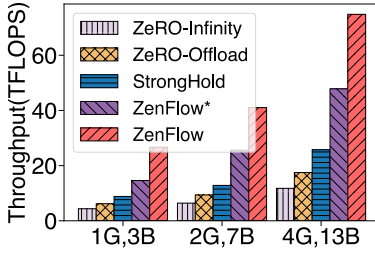


Fig. 11. Throughput across different model and GPU count configurations.

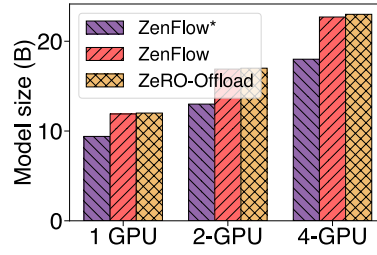


Fig. 12. Maximum model size supported by each system vs. GPU count.

ZenFlow Hyperparameters. ZenFlow introduces two additional hyperparameters: the update interval S and the importance selection ratio $\text{topk_ratio } k$. Unless otherwise specified, we set $S=4$, meaning less important parameters are updated once every 4 iterations and $k=0.01$, meaning top-1% gradients are selected and kept updating on GPU (see §3.2). We provide a detailed analysis of these hyperparameter choices and their impact on accuracy and performance in §6.11.

6.2 End-to-end Training Efficacy

Training Throughput. We first evaluate the end-to-end training throughput of ZenFlow and compare it with state-of-the-art offloading baselines described above. We adopt representative fine-tuning configurations across different model scales: Qwen2.5-3B on 1 GPU with a batch size of 64, Llama2-7B on 2 GPUs with a batch size of 64, and Llama2-13B on 4 GPUs with a batch size of 48. Throughput is reported in TFLOPS using the DeepSpeed FLOPS profiler [17].

ZeRO-Infinity consistently exhibits the lowest throughput due to communication overhead incurred by ZeRO Stage 3 [49]. Therefore, in subsequent tests, we focus on baselines configured with ZeRO Stage 2, which achieve higher GPU utilization and more clearly highlight the impact of GPU stalls on end-to-end training. StrongHold improves over ZeRO-Offload by overlapping part of the GPU computation (i.e., backward pass) with CPU updates; however, the optimizer update phase remains dominant and limits its speedup. ZenFlow* introduces gradient decoupling based on importance, enabling notable performance gains. ZenFlow further improves efficiency by aggressively overlapping CPU updates within the accumulation phase. As shown in Fig. 11, ZenFlow consistently outperforms all baselines across all configurations, achieving on average $4.3\times$ speedup over ZeRO-Offload and $6.3\times$ speedup over ZeRO-Infinity.

Model Scale. We compare the maximum trainable model size under different systems. As shown in Fig. 12, both ZenFlow and ZeRO-Offload offload only optimizer states to ensure a fair comparison. ZenFlow achieves comparable model scalability across 1, 2, and 4 GPUs. ZenFlow* supports slightly smaller models due to additional GPU memory overhead incurred by maintaining a dedicated optimizer for important gradients.

Accuracy and Speedup. Next, we evaluate ZenFlow and baseline methods on four representative GLUE tasks (MNLI, QNLI, QQP, and SST-2) across a range of model sizes and families. As shown in Fig. 13, ZenFlow achieves competitive or superior accuracy compared to existing offloading methods.

Notably, for larger models such as Llama2-7B and Llama2-13B, ZenFlow consistently outperforms baselines. This improvement comes from our importance-aware design: By selectively prioritizing important channels during fine-tuning, ZenFlow preserves the essential learning capacity of the model with significant speedup gains.

In some cases, such as with the smaller OPT-350M model, ZenFlow yields slightly lower accuracy due to the fixed update interval setting ($S = 4$), which may be too coarse to capture rapid gradient changes during early training. This can be addressed via auto-tuning, as discussed in §6.11.

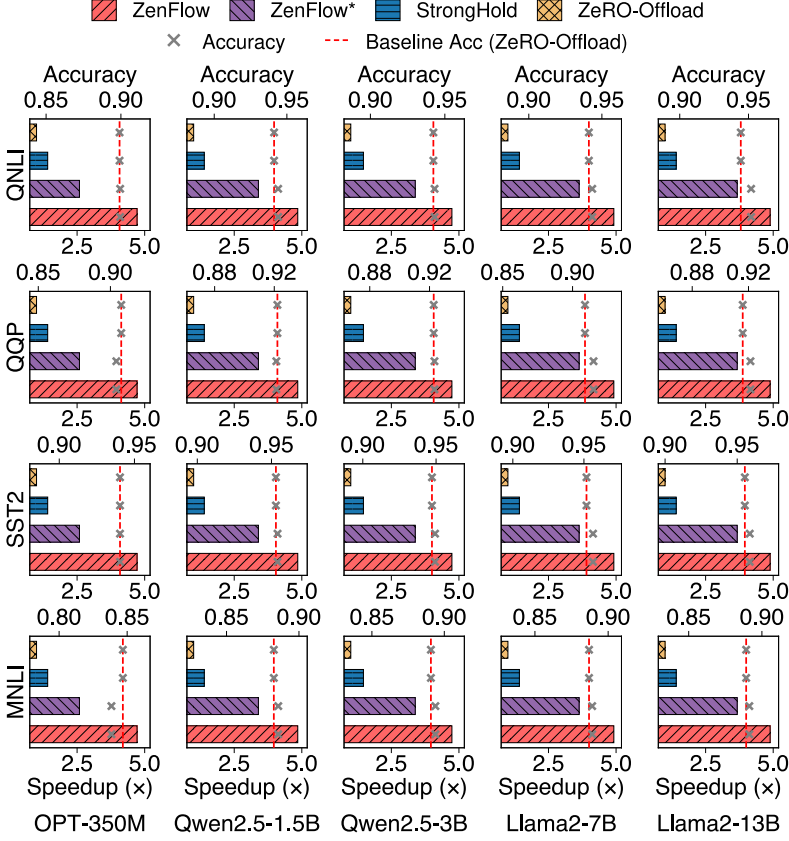


Fig. 13. Accuracy vs. per-iteration speedup on GLUE tasks for various models and baselines. The red dashed line marks the baseline accuracy achieved by ZeRO-Offload. Results close to the red line indicate accuracy comparable to the baseline while achieving significant speedup.

6.3 Time Breakdown and GPU Stall Analysis

We evaluate ZenFlow under three hardware configurations. The first setup reflects practical training environments with $4 \times$ A100 GPUs and full CPU parallelism. The second setup upgrades to $4 \times$ H100 GPUs with the same CPU capacity. The third setting explores the impact of CPU under-provisioning—common in shared clusters where users share the same GPU node but are allocated only a small, exclusive portion of CPU resources (e.g., 8 cores per user).

As shown in Fig. 10, ZenFlow significantly reduces CPU-induced GPU stall time across all settings. By decoupling and overlapping CPU-side updates, ZenFlow consistently eliminates over 80% of GPU stalls, leading to a $2.9 \times$ – $5 \times$ end-to-end speedup compared to ZeRO-Offload. For larger models (e.g., Llama2-7B and 13B), the benefits are more pronounced: the CPU-side overhead becomes a bottleneck due to heavy optimizer updates, even with highly parallelized, AVX-optimized CPUAdam.

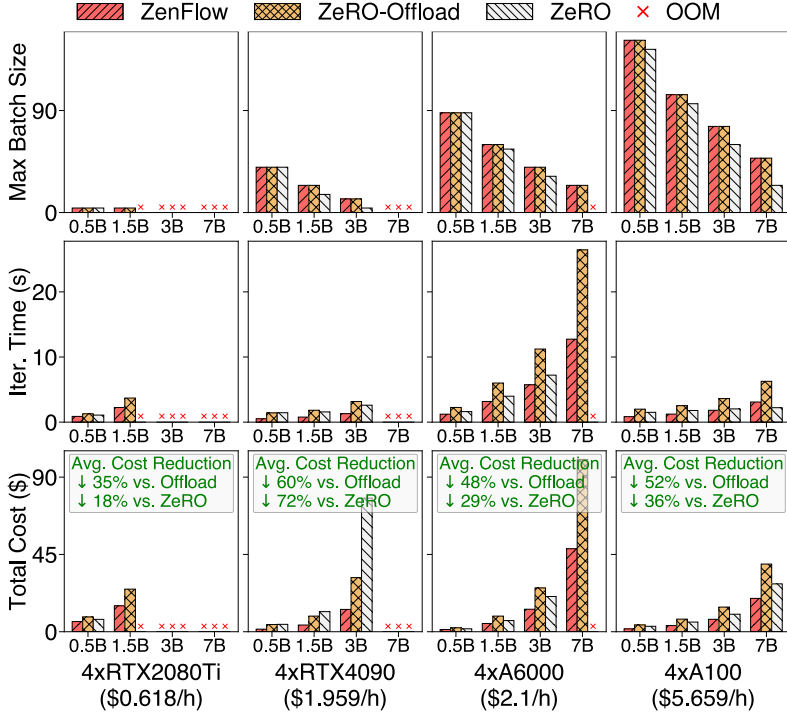


Fig. 14. Cost comparison for fine-tuning Qwen (0.5B–7B) on Alpaca-52K with manually tuned maximum batch sizes. We report the maximum feasible batch size (top), per-iteration time (middle), and total end-to-end training cost (bottom) across different GPU platforms. ZenFlow enables lower training cost by (i) supporting larger batch sizes on cheaper hardware through offloading, and (ii) maintaining competitive iteration speed despite CPU-GPU heterogeneity. As a result, ZenFlow achieves substantial reductions in end-to-end training cost compared to ZeRO-Offload and GPU-only ZeRO, while scaling to larger models without OOM.

ZenFlow effectively mitigates this bottleneck, reducing the CPU:GPU compute time ratio from over 12 : 1 (in baseline runs on 7B / 13B models) to 1 : 1 or lower.

While ZenFlow introduces minor I/O overhead due to gradient gathering, these costs are effectively hidden by lightweight gradient selection and pipelined execution. The detailed breakdown of this overhead is presented in §6.12.

6.4 Practical Cost Savings

Using RunPod on-demand pricing [54], we compare ZenFlow with ZeRO-Offload and GPU-only ZeRO, where GPU-only ZeRO is the full-GPU-memory baseline without CPU offloading. As shown in Fig. 14, ZenFlow extends the model scale beyond what GPU ZeRO-only can support (e.g., 7B parameters on A6000), enabling fine-tuning of larger models without using more expensive hardware. In addition, ZenFlow improves efficiency: by leveraging offloading to support larger batch sizes and overlap updates, ZenFlow achieves higher end-to-end throughput than both ZeRO and ZeRO-Offload. This results in faster completion of the Alpaca52K fine-tuning task and lower overall cost. Even when the optimizer overhead is not fully amortized (e.g., 7B on A100), the larger batch sizes enabled by ZenFlow still reduce cost compared to traditional GPU training. Overall, ZenFlow reduces cost by 35–60% compared to ZeRO-Offload and by 18–72% compared to GPU-only

Table 4. Channel-level gradient ℓ_2^2 concentration over 1,000 Alpaca-52K fine-tuning steps. Top-1% coverage equals the concentration factor over uniform. Jaccard is computed between consecutive steps; random baseline ≈ 0.005 .

Model	Opt.	Top-1%	Top-10%	Jaccard
Qwen2.5-7B	AdamW	67.7%	81.6%	0.64
Qwen1.5-MoE-A2.7B	AdamW	54.8%	72.9%	0.54
Qwen2-VL-7B-Instruct	AdamW	62.9%	79.3%	0.72
BART-large	AdamW	68.5%	81.7%	0.40
Qwen2.5-7B	SGD	54.3%	71.4%	0.81
Qwen1.5-MoE-A2.7B	SGD	54.7%	72.3%	0.58
Qwen2-VL-7B-Instruct	SGD	45.5%	77.7%	0.53
BART-large	SGD	51.0%	73.5%	0.66

ZeRO, demonstrating the practical benefits of importance-aware offloaded training. Compared with GPU-only ZeRO, ZenFlow trades limited CPU offloading overhead for larger feasible batches and trainability when the model would otherwise run out of GPU memory.

6.5 Convergence Validation

We evaluate the convergence behavior of ZenFlow compared to ZeRO-Offload on the GLUE benchmark using OPT-350M. As shown in Fig. 15, we report the training loss on four representative GLUE tasks. Across all tasks, ZenFlow exhibits stable and competitive convergence. Its loss curves closely follow those of ZeRO-Offload, often with low variance (e.g., MNLI), converging at a similar rate in terms of iterations, but achieving much faster absolute convergence time due to reduced iteration latency. Results for other models follow similar trends and are omitted here due to space limit.

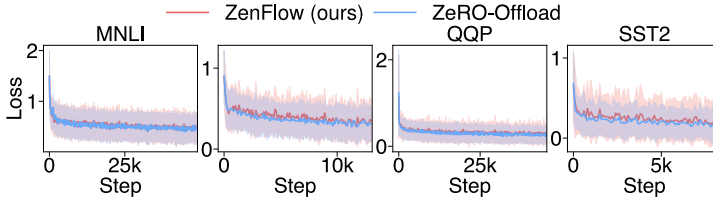


Fig. 15. Convergence on GLUE with OPT-350M. ZenFlow’s loss matches ZeRO-Offload across tasks.

6.6 Gradient Locality Across Architectures

As shown in Table 4, strong channel locality appears across dense decoder-only, MoE, vision-language, and encoder-decoder models under both AdamW and SGD. The top-1% channels capture 45.5–68.5% of gradient ℓ_2^2 , i.e., 45.5–68.5 $\times$ higher concentration than a uniform/random distribution, and their Jaccard similarity across steps is 0.40–0.81 versus a random baseline of ≈ 0.005 .

6.7 Validation on MoE Architecture

We further validate ZenFlow on the Qwen1.5-MoE-A2.7B (14.3B parameters) MoE architecture. On QNLI/MNLI/QQP/SST-2, ZeRO-Offload/ZenFlow achieve 87.45/87.42, 84.93/85.45, 86.27/86.83, and 80.39/81.67, respectively. This comparable accuracy confirms that importance-aware channel selection also applies to sparse expert models.

6.8 PEFT Comparison

Table 5 compares GaLore and ZenFlow on GLUE accuracy across model scales on our A100 testbed. Because GaLore is not compatible with ZeRO-2 sharding and keeps optimizer-state memory, its memory footprint remains high; ZenFlow keeps comparable iteration time while reducing peak memory footprint (e.g., 7B from 53.2GB to 18.5GB) (Table 6).

Table 5. PEFT comparison on GLUE accuracy (%).

Task	System	Qwen2.5-1.5B	Qwen2.5-3B	Llama2-7B
QNLI	GaLore	90.74	91.54	93.30
	ZenFlow	94.40	94.31	94.34
QQP	GaLore	86.82	87.53	85.10
	ZenFlow	92.14	92.33	90.93
SST-2	GaLore	94.95	95.76	96.79
	ZenFlow	95.41	95.61	95.64
MNLI	GaLore	84.00	84.20	87.30
	ZenFlow	88.71	89.52	88.73

Table 6. PEFT comparison on system metrics on the A100 testbed.

System	Metric	Qwen2.5-1.5B	Qwen2.5-3B	Llama2-7B
ZenFlow	Memory footprint	8.9GB	13.2GB	18.5GB
GaLore	Memory footprint	15.5GB	28.4GB	53.2GB
ZenFlow	Avg. iter. time	372ms	673ms	1,322ms
GaLore	Avg. iter. time	451ms	660ms	999ms

6.9 Randomized Selection Ablation

To isolate the effect of importance-aware selection, we compare ZenFlow with two randomized variants on Qwen2.5-3B: random selection keeps a random k fraction of channels on GPU, while random discard drops the remaining gradients without CPU-side accumulation. As shown in Table 7, random selection consistently performs worse than ZenFlow, reducing average accuracy by 4.3 percentage points. This confirms that selecting high-importance channels, rather than merely retaining any channels under the same budget, is critical.

Table 7. Randomized selection ablation on Qwen2.5-3B GLUE accuracy (%).

System	QNLI	QQP	SST-2	MNLI	Avg.
ZenFlow	94.31	92.33	95.61	89.52	92.94
Random discard	93.65	91.70	93.69	88.17	91.80
Random selection	90.77	87.60	90.89	85.39	88.66

6.10 Broader Downstream Task Evaluation

We further evaluate Qwen2.5-3B on MMLU and GSM8K, which measure knowledge and mathematical reasoning capabilities. We report standard zero-shot and few-shot accuracy to assess whether ZenFlow preserves general downstream capability after fine-tuning. As shown in Table 8, ZenFlow achieves comparable accuracy across shot settings, with only small task-dependent variations.

Table 8. Broader downstream accuracy comparison (%).

Task	System	0-shot	5-shot	8-shot	10-shot
MMLU	ZeRO-Offload	50.59	53.71	54.88	59.38
	ZenFlow	50.59	54.10	54.68	59.08
GSM8K	ZeRO-Offload	2.34	63.67	68.36	69.92
	ZenFlow	2.34	63.37	70.80	71.88

Table 9. Warmup-ratio sensitivity on GLUE accuracy (%).

Warmup	QNLI	QQP	SST-2	MNLI	Avg.	Speedup
0%	93.84	92.27	95.89	89.11	92.78	4.97×
5%	94.31	92.33	95.61	89.52	92.94	4.69×
10%	94.32	92.18	95.61	89.78	92.97	4.14×

6.11 Sensitivity of Hyperparameters

We investigate the impact of hyperparameters in ZenFlow, specifically the top- k ratio and the CPU-side update interval. By assigning longer update intervals to less important gradients, ZenFlow further reduces GPU-side stalls. However, this selective staleness can slightly degrade accuracy when training iterations are limited—e.g., using $S=16$ results in a noticeable 0.02 drop in accuracy within just 3 epochs as shown in Fig. 16(a). This tradeoff can be mitigated with extended training iterations, as stale gradients gradually catch up due to ZenFlow’s faster per-iteration execution.

From Fig. 16(a) we observe that ZenFlow is generally robust to variations in the top- k ratio in most cases, as ZenFlow preserves all gradients with bounded staleness. Increasing the update interval typically improves speedup. However, using an overly aggressive update interval (e.g., $S=16$), a smaller ratio may degrade accuracy due to increased staleness penalty (e.g., from 0.881 to 0.873 with 1% top- k). In general, we find that a 1% ratio with $S=4$ strikes a good balance between accuracy and efficiency (see §2.3).

Warmup. Warmup lets ZenFlow perform channel selection after early gradients stabilize. On Qwen2.5-7B, top-1% channel coverage rises from 63.8% early in training to 67.7% later. We vary the warmup ratio on Qwen2.5-3B; as shown in Table 9, 5% warmup provides a good tradeoff, preserving strong speedup while avoiding the QNLI accuracy drop observed without warmup.

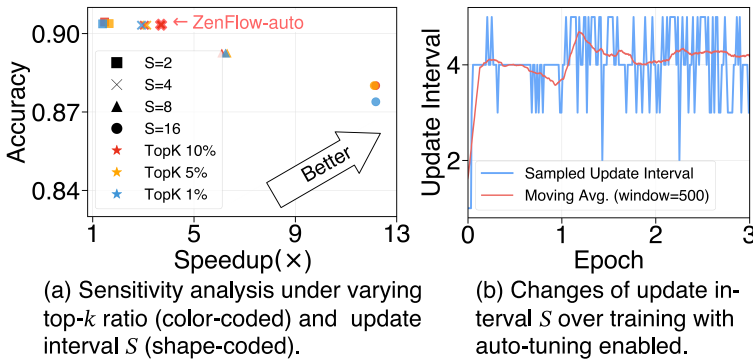


Fig. 16. Sensitivity analysis of key hyperparameters.

To empirically balance accuracy and speedup, we analyze how quickly less important gradients accumulate to match the significance of important ones. For instance, a low-importance gradient starting at 0.1 may grow to 0.5 within five iterations, approaching the magnitude of high-importance gradients. This observation informs our auto-update mechanism shown in Fig. 16(b): early in training, the update interval is kept short (e.g., 1-2), ensuring responsiveness, while later it is adaptively relaxed to 4–5 as training stabilizes. This dynamic configuration delivers higher speedups in later stages without compromising final accuracy, outperforming fixed configurations such as $S=4$.

6.12 Communication Overhead Breakdown

ZenFlow’s lightweight gradient gathering dramatically reduces communication volume and incurs minimal runtime overhead. As shown in Fig. 17, it achieves over 6,000 $\times$ reduction in communication volume and adds less than 0.2s overhead per iteration—substantially lower than full gradient gathering, even on large models like Llama2-13B.

6.13 Applicability under TP and PP

We evaluate ZenFlow with tensor parallelism (TP) and pipeline parallelism (PP) by enabling ZenFlow’s importance-aware offloading workflow in Megatron-LM [58]. Fig. 18 shows ZenFlow* reduces iteration time under both TP=4 and TP=2+PP=2 on 4 $\times$ A100, even for 13B models. By reducing unnecessary compute and I/O overhead for unimportant parameters, ZenFlow’s gains are parallelism-agnostic and hold beyond ZeRO.

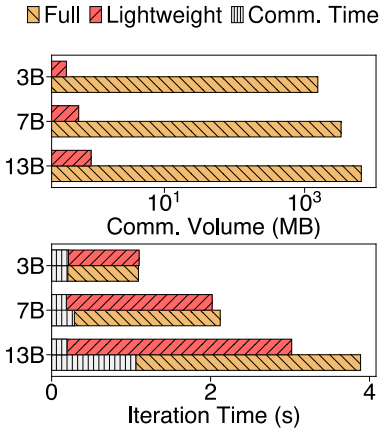


Fig. 17. Gradient gathering communication overhead breakdown.

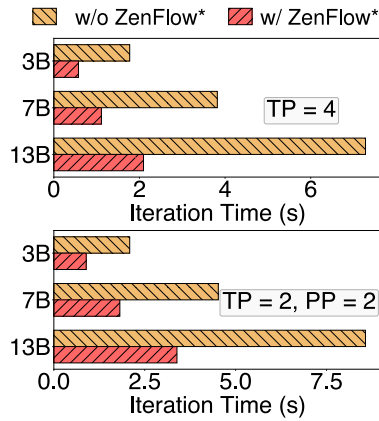


Fig. 18. Iteration time with and without ZenFlow* under TP and TP+PP.

7 Related Work

Tensor Offloading for Training. Prior work [6, 50, 53, 61] offloads tensors to CPU memory to scale training but largely overlooks CPU–GPU pipeline interactions, leaving stalls from CPU latency and PCIe transfers [44]. These systems make large models trainable under GPU memory constraints, yet generally update all parameters uniformly despite the hardware imbalance. StrongHold [61] overlaps layer-wise CPU and GPU computation, but the performance gap between the devices still exposes CPU-induced stalls; overlap alone therefore cannot fully hide slow CPU updates.

Storage-focused systems such as SwapAdvisor [24] and G10 [71] optimize placement across GPU, CPU, and SSD while treating the CPU primarily as storage rather than explicitly coordinating its computation with GPU updates. Recent systems exploit high-bandwidth CPU–GPU superchips for efficient offloading [36, 43], but rely on specialized interconnects such as NVLink-C2C that are unavailable on widely deployed A100 and H100 GPUs. In contrast, ZenFlow is importance- and hardware-aware, decoupling important and non-important updates to minimize stalls.

Gradient Sparsity and Compression. Prior studies show that gradients exhibit sparsity and can be filtered using threshold-based selection [3, 19, 20, 25, 39, 60]; transmitting only top- k or above-threshold gradients can preserve training quality while reducing communication. In offloading settings, this observation motivates prioritizing useful gradients instead of moving every gradient with the same urgency. Smart-Infinity [27] and LSP-Offload [11] reduce offloading I/O through lossy compression, dropping small gradients or using learned projections at the cost of gradient fidelity. ZenFlow instead retains the remaining gradients through CPU-side accumulation while scheduling important updates on the GPU. Compression remains orthogonal to ZenFlow and can further optimize its scheduling pipeline.

Asynchronous Training. Asynchrony can accelerate distributed training [9, 33, 37, 65, 68, 70, 72], but non-deterministic execution introduces stale gradients that can degrade convergence and reproducibility [16, 35, 73, 76]. In typical schemes, workers update shared state according to unpredictable completion times, making the resulting update sequence difficult to reproduce. Prior work therefore bounds staleness [9, 14, 35, 70]. ZenFlow instead uses bounded asynchronous execution specifically to hide CPU-side latency while retaining a predetermined, repeatable update order.

Parameter-Efficient vs. Full-Parameter Fine-Tuning. Parameter-Efficient Fine-Tuning (PEFT) methods (e.g., adapters [21], prefix-tuning [34], LoRA [23], Q-LoRA [18], mLoRA [69], GaLore [74]) reduce fine-tuning cost by updating only a small subset of parameters, significantly lowering memory and optimizer overhead. However, this efficiency comes at a cost: the constrained parameter space can bottleneck the model’s ability to learn from diverse datasets or sustain long-term training [10, 12, 15, 38, 47]. Unlike PEFT, ZenFlow targets full-parameter fine-tuning rather than restricting the trainable parameter space, preserving full-weight flexibility while improving system performance. The two approaches address different bottlenecks: ZenFlow is complementary to PEFT and can be integrated with low-rank methods.

8 Conclusion

ZenFlow decouples fast GPU and slow CPU updates to reduce I/O and GPU stalls in LLM fine-tuning. It overlaps in-place GPU updates of important gradients with CPU accumulation of the remainder. By exploiting gradients’ spatial and temporal locality for scalable importance estimation, ZenFlow accelerates fine-tuning without global synchronization while preserving accuracy and reproducibility. ZenFlow has been integrated into DeepSpeed.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation under Grants OAC-2411009, CNS-2322860, OAC-2403313, and CCF-2316202. We also acknowledge NSF CloudBank, Modal, and Lambda for providing cloud GPU credits, and the Commonwealth Cyber Initiative (CCI), an investment in the advancement of cyber research, innovation, and workforce development. For more information about CCI, visit cyberinitiative.org.

References

- [1] Daniel Adiwardana, Minh-Thang Luong, David R. So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, and Quoc V. Le. 2020. Towards a Human-like Open-Domain Chatbot. *arXiv:2001.09977* [cs.CL] <https://arxiv.org/abs/2001.09977>
- [2] Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. 2020. Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning. *arXiv:2012.13255* [cs.LG] <https://arxiv.org/abs/2012.13255>
- [3] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021* (2017).
- [4] Dan Alistarh, Torsten Hoeftler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cedric Renggli. 2018. The convergence of sparsified gradient methods. In *Advances in Neural Information Processing Systems*, Vol. 31.
- [5] Guangji Bai, Zheng Chai, Chen Ling, Shiyu Wang, Jiaying Lu, Nan Zhang, Tingwei Shi, Ziyang Yu, Mengdan Zhu, Yifei Zhang, Xinyuan Song, Carl Yang, Yue Cheng, and Liang Zhao. 2024. Beyond Efficiency: A Systematic Survey of Resource-Efficient Large Language Models. *arXiv:2401.00625* [cs.LG] <https://arxiv.org/abs/2401.00625>
- [6] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. 2021. Efficient Combination of Rematerialization and Offloading for Training DNNs. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 23844–23857. https://proceedings.neurips.cc/paper_files/paper/2021/file/c8461bf13fca8a2b9912ab2eb1668e4b-Paper.pdf
- [7] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010: 19th International Conference on Computational Statistics Paris France, August 22-27, 2010 Keynote, Invited and Contributed Papers*. Springer, 177–186.
- [8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) (*NIPS '20*). Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.
- [9] Zheng Chai, Yujing Chen, Ali Anwar, Liang Zhao, Yue Cheng, and Huzefa Rangwala. 2021. FedAT: A high-performance and communication-efficient federated learning system with asynchronous tiers. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–16.
- [10] Guanzheng Chen, Fangyu Liu, Zaiqiao Meng, and Shangsong Liang. 2022. Revisiting Parameter-Efficient Tuning: Are We Really There Yet?. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 2612–2626. doi:10.18653/v1/2022.emnlp-main.168
- [11] Siyuan Chen, Zhuofeng Wang, Zelong Guan, Yudong Liu, and Phillip B Gibbons. 2025. Practical Offloading for Fine-Tuning LLM on Commodity GPU via Learned Sparse Projectors. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 23614–23622.
- [12] Eric Nuertey Coleman, Luigi Quarantiello, Ziyue Liu, Qinwen Yang, Samrat Mukherjee, Julio Hurtado, and Vincenzo Lomonaco. 2025. Parameter-Efficient Continual Fine-Tuning: A Survey. *arXiv preprint arXiv:2504.13822* (April 2025). Version 2, August 2025. <https://arxiv.org/abs/2504.13822>
- [13] Ben Cottier, Robi Rahman, Loredana Fattorini, Nestor Maslej, Tamay Besiroglu, and David Owen. 2025. The rising costs of training frontier AI models. *arXiv:2405.21015* [cs.CY] <https://arxiv.org/abs/2405.21015>
- [14] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2014. Exploiting bounded staleness to speed up big data analytics. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) (*USENIX ATC'14*). USENIX Association, USA, 37–48.
- [15] Gaole Dai, Chun-Kai Fan, Yiming Tang, Qizhe Zhang, Yulu Gan, Zhi Zhang, Chengqing Zeng, Shanghang Zhang, and Tiejun Huang. 2024. Discovering Long-Term Effects on Parameter Efficient Fine-tuning. *arXiv preprint arXiv:2409.06706* (September 2024). Submitted to ICLR 2025. <https://arxiv.org/abs/2409.06706>
- [16] Wei Dai, Yi Zhou, Nanqing Dong, Hao Zhang, and Eric P Xing. 2018. Toward understanding the impact of staleness in distributed machine learning. *arXiv preprint arXiv:1810.03264* (2018).
- [17] DeepSpeed Team. 2025. DeepSpeed Flops Profiler. <https://www.deepspeed.ai/tutorials/flops-profiler/>.
- [18] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems* 36 (2023), 10088–10115.
- [19] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. 2016. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on machine learning in hpc environments (MLHPC)*. IEEE, 1–8.

- [20] Gongfan Fang, Hongxu Yin, Saurav Muralidharan, Greg Heinrich, Jeff Pool, Jan Kautz, Pavlo Molchanov, and Xinchao Wang. 2024. MaskLLM: learnable semi-structured sparsity for large language models. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) (NIPS '24). Curran Associates Inc., Red Hook, NY, USA, Article 248, 23 pages.
- [21] Shwai He, Liang Ding, Daize Dong, Miao Zhang, and Dacheng Tao. 2022. Sparseadapter: An easy approach for improving the parameter-efficiency of adapters. *arXiv preprint arXiv:2210.04284* (2022).
- [22] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556* (2022).
- [23] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.
- [24] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. 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, 1341–1355. doi:10.1145/3373376.3378530
- [25] Weiyu Huang, Yuezhou Hu, Jun Zhu, and Jianfei Chen. 2025. CAST: Continuous and Differentiable Semi-Structured Sparsity-Aware Training for Large Language Models. arXiv:2509.25996 [cs.LG] <https://arxiv.org/abs/2509.25996>
- [26] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf
- [27] Hongsun Jang, Jaeyong Song, Jaewon Jung, Jaeyoung Park, Youngsok Kim, and Jinho Lee. 2024. Smart-Infinity: Fast Large Language Model Training using Near-Storage Processing on a Real System. arXiv:2403.06664 [cs.AR] <https://arxiv.org/abs/2403.06664>
- [28] Sebastian Jaszczur, Aakanksha Chowdhery, Afroz Mohiuddin, Lukasz Kaiser, Wojciech Gajewski, Henryk Michalewski, and Jonni Kanerva. 2021. Sparse is enough in scaling transformers. *Advances in Neural Information Processing Systems* 34 (2021), 9895–9907.
- [29] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).
- [30] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. arXiv:2004.04906 [cs.CL] <https://arxiv.org/abs/2004.04906>
- [31] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG] <https://arxiv.org/abs/1412.6980>
- [32] Anastasia Koloskova, Sebastian U Stich, and Martin Jaggi. 2022. Sharper convergence guarantees for asynchronous SGD for distributed and federated learning. *Advances in Neural Information Processing Systems* 35 (2022), 17202–17215.
- [33] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) (OSDI'14). USENIX Association, USA, 583–598.
- [34] Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190* (2021).
- [35] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. 2015. Asynchronous parallel stochastic gradient for nonconvex optimization. *Advances in neural information processing systems* 28 (2015).
- [36] Xinyu Lian, Masahiro Tanaka, Olatunji Ruwase, and Minjia Zhang. 2025. SuperOffload: Unleashing the Power of Large-Scale LLM Training on Superchips. arXiv:2509.21271 [cs.LG] <https://arxiv.org/abs/2509.21271>
- [37] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. 2018. Asynchronous decentralized parallel stochastic gradient descent. In *International conference on machine learning*. PMLR, 3043–3052.
- [38] Baohao Liao, Yan Meng, and Christof Monz. 2023. Parameter-Efficient Fine-Tuning without Introducing New Latency. 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, 4242–4260. doi:10.18653/v1/2023.acl-long.233
- [39] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).
- [40] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

- [41] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=Bkg6RiCqY7>
- [42] Kairong Luo, Haodong Wen, Shengding Hu, Zhenbo Sun, Zhiyuan Liu, Maosong Sun, Kaifeng Lyu, and Wenguang Chen. 2025. A Multi-Power Law for Loss Curve Prediction Across Learning Rate Schedules. *arXiv preprint arXiv:2503.12811* (2025).
- [43] Shutian Luo, Ali Zafar Sadiq, Rui Yang, Mingye Zhang, Haiying Shen, Wei Wang, and Yue Cheng. 2026. C2CServe: Leveraging NVLink-C2C for Elastic Serverless LLM Serving on MIG. *arXiv:2605.19481 [cs.OS]* <https://arxiv.org/abs/2605.19481>
- [44] Avinash Maurya, Jie Ye, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. Breaking the Memory Wall: A Study of I/O Patterns and GPU Memory Utilization for Hybrid CPU-GPU Offloaded Optimizers. In *Proceedings of the 14th Workshop on AI and Scientific Computing at Scale Using Flexible Computing Infrastructures* (Pisa, Italy) (*FlexScience'24*). Association for Computing Machinery, New York, NY, USA, 9–16. doi:10.1145/3659995.3660038
- [45] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) (*SOSP '19*). Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3341301.3359646
- [46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv:1912.01703 [cs.LG]* <https://arxiv.org/abs/1912.01703>
- [47] Pranav Patel. 2025. Guide to fine-tuning LLMs using PEFT and LoRA techniques. Mercity Blog. <https://www.mercity.ai/blog-post/fine-tuning-llms-using-peft-and-lora>
- [48] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).
- [49] 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.
- [50] 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. doi:10.1145/3458817.3476205
- [51] 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.
- [52] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. 2021. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 598–611. doi:10.1109/HPCA51647.2021.00057
- [53] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.
- [54] RunPod. [n. d.]. RunPod Pricing. <https://www.runpod.io/pricing>.
- [55] Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive learning rates with sublinear memory cost. In *Proceedings of the 35th International Conference on Machine Learning*.
- [56] Li Shen, Yan Sun, Zhiyuan Yu, Liang Ding, Xinmei Tian, and Dacheng Tao. 2023. On Efficient Training of Large-Scale Deep Learning Models: A Literature Review. *arXiv:2304.03589 [cs.LG]* <https://arxiv.org/abs/2304.03589>
- [57] Sheng Shen, Zhewei Yao, Amir Gholami, Michael Mahoney, and Kurt Keutzer. 2020. PowerNorm: Rethinking batch normalization in transformers. In *International conference on machine learning*. PMLR, 8741–8751.
- [58] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-Lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).
- [59] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. 2018. Sparsified SGD with memory. In *Advances in Neural Information Processing Systems*, Vol. 31.
- [60] Nikko Ström. 2015. Scalable distributed DNN training using commodity GPU cloud computing. (2015).
- [61] Xiaoyang Sun, Wei Wang, Shenghao Qiu, Renyu Yang, Songfang Huang, Jie Xu, and Zheng Wang. 2022. Stronghold: fast and affordable billion-scale deep learning model training. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–17.

- [62] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] <https://arxiv.org/abs/2302.13971>
- [63] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL] <https://arxiv.org/abs/2307.09288>
- [64] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. arXiv:1804.07461 [cs.CL] <https://arxiv.org/abs/1804.07461>
- [65] Qinlong Wang, Tingfeng Lan, Yinghao Tang, Bo Sang, Ziling Huang, Yiheng Du, Haitao Zhang, Jian Sha, Hui Lu, Yuanchun Zhou, Ke Zhang, and Mingjie Tang. 2024. DLRouter-RM: Resource Optimization for Deep Recommendation Models Training in the Cloud. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4130–4144. doi:10.14778/3685800.3685832
- [66] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. arXiv:2109.00859 [cs.CL] <https://arxiv.org/abs/2109.00859>
- [67] Zirui Wang, Tingfeng Lan, Zhaoyuan Su, Juncheng Yang, and Yue Cheng. 2025. Towards Efficient LLM Storage Reduction via Tensor Deduplication and Delta Compression. arXiv:2505.06252 [cs.DB] <https://arxiv.org/abs/2505.06252>
- [68] Eric P Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A new platform for distributed machine learning on big data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1335–1344.
- [69] Zhengmao Ye, Dengchun Li, Zetao Hu, Tingfeng Lan, Jian Sha, Shicong Zhang, Lei Duan, Jie Zuo, Hui Lu, Yuanchun Zhou, and Mingjie Tang. 2025. mLoRA: Fine-Tuning LoRA Adapters via Highly-Efficient Pipeline Parallelism in Multiple GPUs. *Proc. VLDB Endow.* 18, 6 (Feb. 2025), 1948–1961. doi:10.14778/3725688.3725718
- [70] Hanfei Yu, Hao Wang, Devesh Tiwari, Jian Li, and Seung-Jong Park. 2024. Stellaris: Staleness-Aware Distributed Reinforcement Learning with Serverless Computing. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–17.
- [71] Haoyang Zhang, Yirui Zhou, Yuqi Xue, Yiqi Liu, and Jian Huang. 2023. G10: Enabling an efficient unified gpu memory and storage architecture with smart tensor migrations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 395–410.
- [72] Sixin Zhang, Anna E Choromanska, and Yann LeCun. 2015. Deep learning with elastic averaging SGD. *Advances in neural information processing systems* 28 (2015).
- [73] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. 2015. Staleness-aware async-sgd for distributed deep learning. *arXiv preprint arXiv:1511.05950* (2015).
- [74] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. 2024. GaLore: Memory-Efficient LLM Training by Gradient Low-Rank Projection. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*, Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (Eds.). PMLR, 61121–61143. <https://proceedings.mlr.press/v235/zhao24s.html>
- [75] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277* (2023).
- [76] Yi Zhou, Yaoliang Yu, Wei Dai, Yingbin Liang, and Eric Xing. 2016. On convergence of model parallel proximal gradient algorithm for stale synchronous parallel system. In *Artificial Intelligence and Statistics*. PMLR, 713–722.

Received 17 January 2026; revised 19 May 2026; accepted 12 June 2026