ZenFlow: Enabling Stall-Free Offloading Training via Asynchronous Updates

Tingfeng Lan
University of Virginia

Yusen Wu
University of Virginia

Bin Ma
University of California, Merced

Zhaoyuan Su
University of Virginia

Rui Yang
University of Virginia

Tekin Bicer
Argonne National Laboratory

Dong Li
University of California, Merced

Yue Cheng
University of Virginia

Abstract

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 offloading framework that prioritizes important parameters and decouples updates between GPU and CPU. ZenFlow performs in-place updates of important gradients on GPU, while asynchronously offloading and accumulating less important ones on CPU, fulling overlapping CPU work with GPU computation.

To scale across GPUs, ZenFlow introduces a lightweight gradient selection method that exploits a novel spatial and temporal locality property of important gradients, avoid costly global synchronization. ZenFlow achieves up to 5× end-to-end speedup, 2× lower PCIe traffic, and reduces GPU stalls by over 85%, all while preserving accuracy.

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, 20, 32, 47]. While pretrained models [6, 32, 44, 45], offer general-purpose capabilities, fine-tuning them on domain-specific data is often essential for achieving high performance on downstream tasks. However, as LLMs grow to tens or hundreds of billions of parameters, the memory demands of fine-tuning far exceed the capacity of a single GPU, making large-scale training increasingly challenging [9, 39].

To address this memory bottleneck, offloading-based training systems, such as ZeRO-Offload [37] and ZeRO-Infinity [34], have emerged as promising solutions. These systems reduce GPU memory consumption by offloading model states (e.g., gradients, optimizer states) to CPU memory or NVMe SSD. Unfortunately, this comes at the cost of substantial training overhead, as CPU-side updates are orders of magnitude slower than GPU computation, and communication over

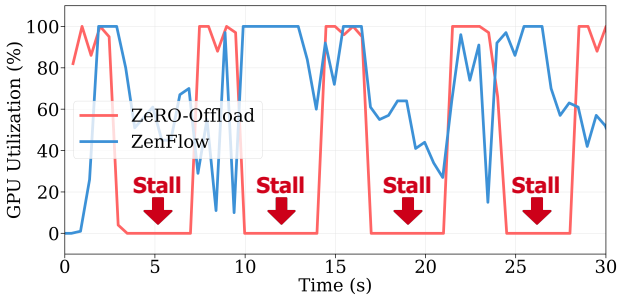


Figure 1. GPU utilization of ZeRO-Offload and ZenFlow for fine-tuning Llama2-7B on 4× A100.

PCIe is constrained by limited bandwidth. For example, on fine-tuning a Llama-2-7B [45] model with 4 A100 GPUs, one training step time experiences a dramatic increase from 0.5s to 7s when enabling offloading, a 14× slowdown.

We identify two major sources of inefficiency in existing offloading systems: (1) *Long stalls caused by 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 CPU with slow computation and limited PCIe bandwidth for updating and transferring the entire model states. In practice, this causes slow CPU to bottleneck fast and more expensive GPU. As shown in Fig. 1, during fie 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.

Our analysis and former study [2, 3, 17, 25] reveal that *LLM gradients are highly imbalanced*: the top 1% of gradients account for over 90% of the total gradient norm. This hardware-agnostic treatment of gradients leads to wasted

computation and communication on low-impact updates, stalling overall training throughput.

We introduce ZenFlow, the **first** offloading system that prioritizes and decouples gradient updates based on both hardware heterogeneity and learning dynamics. 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 high-impact parameters. Meanwhile, remaining less important gradients are offloaded to the **slower** CPU, where they are asynchronously 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. 1, 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 the GPU. This becomes particularly difficult in distributed training settings such as ZeRO [33], 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 spatial and 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 to minimize GPU stalls and I/O overhead.
- We prototype ZenFlow on DeepSpeed [35] 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 the same level of accuracy across diverse LLMs and fine-tuning tasks.

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 [6, 44, 45]. 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 propages 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 [5], Adam [21]).

Distributed Parallel Training with State Sharding. To train large models efficiently, distributed parallelism is widely adopted. When models fit in GPU memory, data parallelism [23] is commonly used, replicating the model across devices and distributing input batches. For larger models that exceed memory capacity, model parallelism [41] and pipeline parallelism [16, 30] partition model layers across devices to utilize aggregate memory.

Traditional data parallelism replicates full model states (parameters, gradients, optimizer states) on each GPU, incurring high memory overhead. To address this, *state sharding* [33] partitions model states across devices, allowing each GPU to manage only a shard. Full states are reconstructed as needed via collective communication. This technique is central to modern large-scale training systems, including DeepSpeed ZeRO [35, 37], Megatron-LM [41], and PyTorch FSDP [53], with DeepSpeed ZeRO being the most representative.

2.2 Memory Offloading

Training Memory Breakdown. Training deep learning models requires memory for parameters, gradients, optimizer states, and activations. Activations are temporary values used during the backward pass (BP), while gradients and optimizer states (e.g., momentum and variance in AdamW) are needed for parameter updates. Among these, parameters, gradients, and optimizer states dominate memory usage and scale linearly with model size. In half precision (BF16/FP16), each parameter takes 2 bytes. Let M denote the total size of parameters; gradients require another M, and optimizer states add $2M$, leading to a total memory footprint of $4M$. As shown in Table 1, fine-tuning Llama2-7B requires 14GB each for parameters and gradients, and 28GB for optimizer states—totaling 56GB, which exceeds the 40GB memory limit of a single A100 GPU.

Offloaded Training. To mitigate the GPU memory bottleneck, some offloading techniques [15, 34, 36, 37] 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

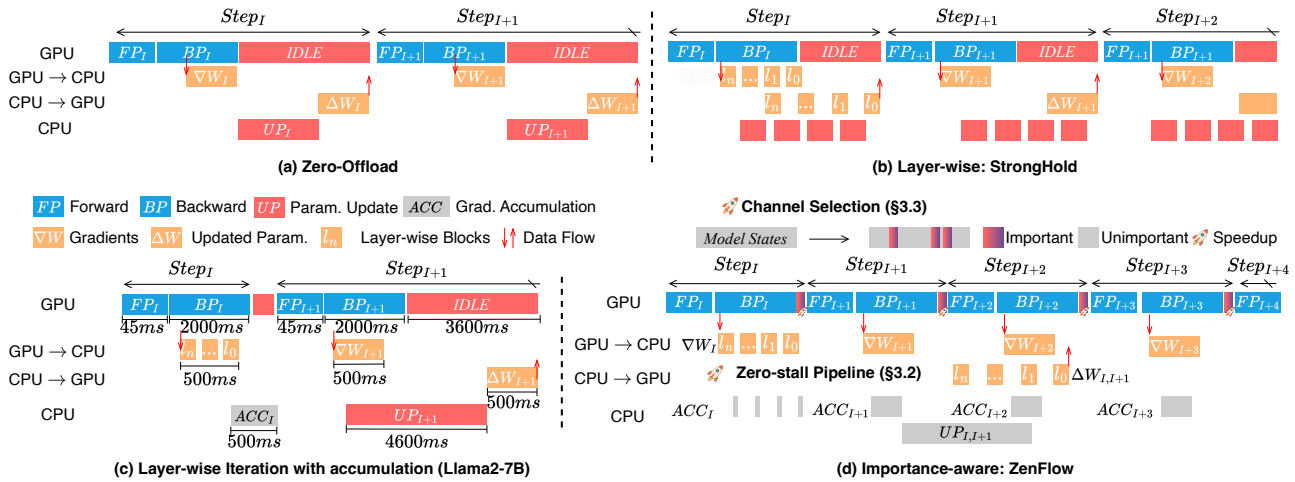


Figure 2. Offloading strategy comparison. (a) **ZeRO-Offload** [37] sequentially executes FP and BP on GPU, then offloads gradients and performs UP on CPU, leaving the GPU idle. (b) **StrongHold** [43] overlaps CPU updates with GPU backward computation and gradient offloading by offloading and updating gradients layer by layer. However, CPU-side update is still too slow to hide, causing GPU stalls. (c) **Example: iteration with accumulation:** Updates occur every two steps—one for accumulation (i.e., gradients are summed without applying updates) (ACC), one for normal update (UP). (d) **ZenFlow** (ours) 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).

memory consumption and enable "out-of-core" training of larger models. ZeRO-Offload [37] in the DeepSpeed ZeRO series [33, 34, 37], is one of those state-of-the-art systems.

Fig. 2 (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 arrive from the CPU.

2.3 Problems and Insights

Problem #1: Fast GPU execution is frequently stalled by slow CPU-side updates. 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. 3). 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

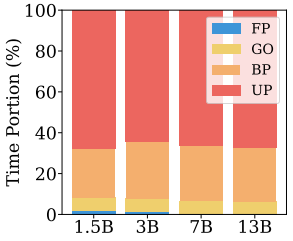


Figure 3. Per-iteration time breakdown under DeepSpeed ZeRO-Offload when training Qwen2.5-{1.5B,3B} and Llama2-{7B, 13B} models with 4 A100 40GB GPUs and a AMD EPYC processor with 64 CPU cores (128 threads, SMT enabled). Gradient offloading (GO) represents the time spent transferring gradients from GPU to CPU.

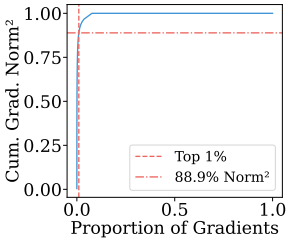


Figure 4. CDF of gradient norm squared across all gradients on fine-tuning Qwen2.5-0.5B on the Alpaca52K dataset. The gradient norm measures the magnitude of each gradient. The top 1% of gradients account for 88.9% of the total gradient norm squared, indicating that a small subset of gradients dominates the parameter update.

update is a significant bottleneck (see Fig. 3). For example, when training a Llama2-7B model, the update stage on the CPU takes approximately 4,600ms—over twice as long as the backward pass time of 2,000ms, 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 [43] is to overlap CPU updates with GPU computation using a strawman method such as layer-wise scheduling (Fig. 2(b)). In this strategy, each layer's gradients are

Table 1. Resource requirement for fine-tuning Llama2-7B (BF16) on 4× A100 80GB GPUs with DeepSpeed ZeRO-Offload.

Memory	**Param.** 14GB (M)	**Optim. States** 28GB ($2M$)	**Gradient** 14GB (M)
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 (M)	**Param. Update** CPU→GPU 14GB (M)

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 parameter 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 to complete (i.e., for all remaining layers from l_{n-1} to l_0). This allows updated parameters to be uploaded back to the GPU earlier, reducing GPU idle time before the next iteration.

Although the layer-wise scheduling can partially overlap CPU and GPU tasks, the CPU update phase is too long to be fully hidden (4,600ms on the CPU vs. 2,000ms on the GPU). As a result, GPU execution is frequently stalled, waiting for the CPU update even with aggressive multi-threaded optimization.

Problem #2: Limited PCIe bandwidth creates a communication bottleneck between CPU and GPU, stalling GPU execution. Offloading the optimizer states to the CPU memory introduces substantial communication overhead. Taking Llama2-7B as an example,

each iteration involves transferring 14GB of gradients from the GPU to the CPU, followed by transferring 14GB of updated parameters from the CPU to the GPU—equivalent to one full model size (M) in each direction (see Fig. 2(c) and Table 1).

Over PCIe 4.0 ×16 with a theoretical bandwidth of 32 GB/s and a throughput of ~28 GB/s, each transfer takes approximately 500ms, resulting in a total of ~1,000ms of I/O overhead per iteration.

Even with an ideal condition where the GPU backward pass (2,000ms) overlaps with the gradient offloading (500ms), parameter update on the CPU (4,600ms), and the transfer of updated parameters from CPU to GPU takes 500ms and the GPU stall remains significant. The total stall time per iteration is calculated as $4,600 + 2 * 500 - 2,000 = 3,600ms$. Fig. 2(c) depicts the details.

In conclusion, the GPU is largely idle, even if we maximize the overlap between the parameter update on the CPU and the backward pass on the GPU.

The root cause for the low GPU utilization is two fold: the CPU updates are inherently slow even when parallelized, and the limited PCIe bandwidth just cannot fully hide the transfer cost. These findings highlights the need of rethinking the CPU-GPU update pipeline. Reducing communication volume and minimizing synchronization between the GPU and CPU are critical to improving training efficiency.

Insight #1: Gradients/parameters with different importance should be treated differently to decouple GPU/CPU execution and reduce communication. It has been widely observed that the gradients of different parameters have different importance in deep neural network (DNN) training [3, 13, 25, 42]. For example, Fig. 4 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. Important gradient updates—those critical to learning—are delayed by slow CPU-side updates, and are forced to wait alongside less important ones.

As introduced earlier in this section, GPU execution is often stalled by the slow CPU update stage, and the GPU must wait for all updated parameters before starting the next iteration. Notably, this includes parameters updated using unimportant gradients, which provide limited benefit but incur CPU-side delay and I/O overhead.

This observation leads to a key research question: *Can we decouple the handling of 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 *asynchronously* and is typically fast enough (e.g., ~500ms) to be hidden within the backward pass, thereby avoiding additional delay and improving GPU utilization.

Problem #3: Full gradient view is expensive in fully sharded distributed training. To leverage gradient importance during training, selecting important gradients is a critical step.

A simple yet effective approach is the top-k selection, which retains the gradients with the highest magnitudes. This strategy has been widely studied and applied in prior work [3, 17]. 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 challenging, because constructing a fully global view would require gathering and synchronizing gradients across

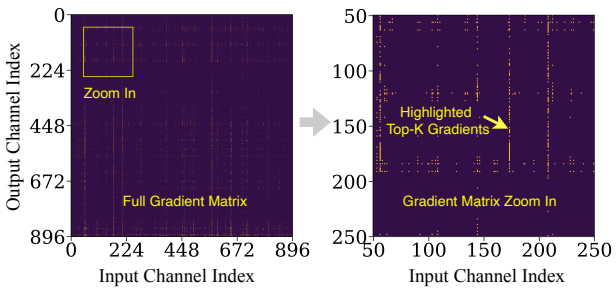


Figure 5. Gradient heatmap during fine-tuning. **Left:** A snapshot of full gradient matrix observed during fine-tuning, where rows and columns represent the two dimensions of the gradient matrix. The X-axis corresponds to the input dimension, with each column capturing gradients associated with a particular input feature (i.e. input channel). The Y-axis (row axis) corresponds to the output dimension. The top-1% largest gradients (by magnitude) are highlighted in orange. **Right:** A zoom-in of a small region from the left figure, clearly showing high-magnitude gradients are concentrated along specific columns (input channels). This aligns with the nature of fine-tuning, where training primarily focuses on a task-specific subset of input features, highlighting strong locality.

all devices, incurring significant communication overhead and causing peak memory usage spikes due to gathered full gradient matrix. Constructing a fully global view hurts the scalability and efficiency of fully sharded training. Therefore, applying the global top-k selection directly is not practical.

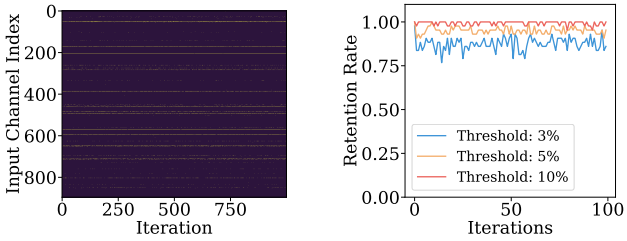
Insight #2: Important LLM gradients show both spatial and temporal locality. To address the challenge posed by the lack of a global gradient view in fully sharded distributed training, we next investigate an important research question: *Do important gradients in LLM fine-tuning exhibit spatial or temporal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

First, we observe clear **spatial locality** in the gradient distribution. As shown in Fig. 5, 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 as further illustrated in Fig. 6a and discussed next.

This result suggests that a small set of input features consistently receive large updates during finetuning. This aligns with prior observations that transformer activations are often localized across channels during inference [18, 40], and such patterns propagate backward to the gradients.

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.



(a) Top-1% input channel timeline. **(b)** Retention rate over iterations.

Figure 6. Temporal locality of important gradients. **(a)** Input channels tracked over 1,000 iterations. The Y-axis shows input channel index, with the top-1% important channels highlighted in yellow. **(b)** Retention rate of top-1% gradients over time when tracking a fixed set of top-k% important channels.

Second, we find that these important input channels also exhibit **temporal locality**. As shown in Fig. 6a, the top 1% of important input channel indices remain stable across iterations, forming persistent horizontal bands over time. This indicates that same small subset of input channels consistently receive large updates during fine-tuning. In other words, these channels contribute more to the model's 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. To quantify this, Fig. 6b reports the retention rate: the fraction of top-1% gradients captured by a fixed set of top-k% input channels. Tracking only the top 10% most important channels (colored in red) retains over 95% of the top-1% gradients across 100 iterations. Even with a narrower 5% threshold (colored in yellow), the retention rate remains consistently above 90%, confirming that the gradient importance is not only spatially concentrated, but also temporally stable throughout fine-tuning.

This spatial and 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 the design of ZenFlow, a system that leverages gradient importance and gradients' spatio-temporal locality to mitigate the I/O bottleneck and GPU execution stalls in offloading training. ZenFlow decouples gradient updates across heterogeneous hardware and reduces communication overhead without sacrificing accuracy.

In this section, we presents the design of ZenFlow, guided by the following goals:

- **Goal #1.** Minimize GPU stalls caused by slow CPU.
- **Goal #2.** Reduce I/O and computation overhead associated with unimportant gradients and parameters.

• **Goal #3.** Preserve accuracy by ensuring no loss of important information.

3.1 Asynchronous 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 section, we assume that an important subset of gradients and their corresponding parameters has already been identified. The mechanism for selecting this subset is described later in §3.3. Here, we focus on how ZenFlow manages the asynchronous offloading and update process once the selection is in place.

As shown in Fig. 2 (d), ZenFlow assigns the important gradient and parameters (highlighted in red and purple-ish color) 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 the corresponding parameter subset, 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.

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. This CPU-side update runs *asynchronously* and is carefully scheduled, so that it perfectly overlaps with GPU computation, avoiding any extra stalls in the training loop (see §3.2). In effect, ZenFlow updates important parameters more frequently using *fast* GPU, while updating less important ones less often on *slow* CPU, but with fully accumulated gradient information.

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

3.2 Zero-stall Pipeline

While the asynchronous 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. In this section, we 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 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 $2M$.

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 N 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 configurations of the accumulation rounds from our analysis for hiding CPU-side updates we will dicuss shortly), and $k = 0.1$ as an example, the average I/O traffic per iteration becomes $1.125M$, which is nearly a 2× reduction compared to DeepSpeed ZeRO-Offload.

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

Critically, we observe that unimportant gradients usually require 4-6 accumulation steps before crossing the importance threshold for update (see §5.5). 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 §3.4 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

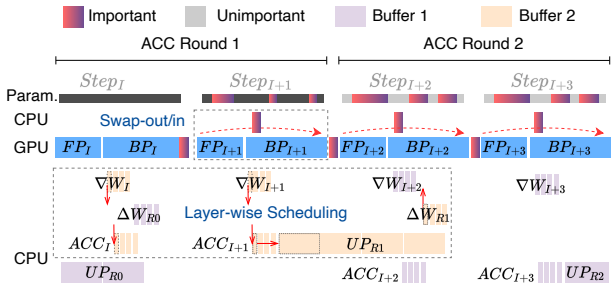


Figure 7. Zero-stall pipeline with double buffering.

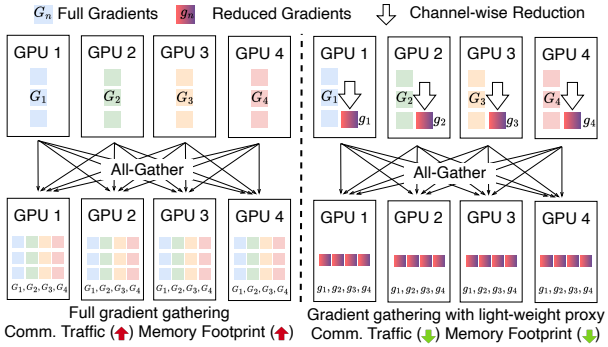


Figure 8. Gradient gathering strategies.

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 $Step_{I+1}$.)

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.

Hyperparameter Auto-tuning. To further improve accuracy under dynamic training conditions, we design Zen-auto, which adaptively tunes the update interval based on observed learning dynamics. Specifically, Zen-automonitors gradient changes across GPUs using a lightweight coordination proxy (detailed in §3.3). For unimportant gradient part, Zen-autotracks 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, Zen-auto immediately triggers its CPU-side update, ensuring timely parameter refresh and stable convergence.

Swapping out/in and Layer-wise Scheduling. To avoid excessive GPU memory consumption from the GPU-side *selective-optimizer*, ZenFlow swap out its optimizer states to CPU and swap 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.

3.3 Gradient Selection

We now turn to the *key question of 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 (i.e., high gradient norms), as they typically contribute more to learning. However, in fully sharded training, computing and comparing all gradient norms globally is prohibitively expensive. To make this feasible in distributed training, ZenFlow introduces a lightweight yet effective approximation that leverages the spatial and 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 `AllGather`, which is expensive in distributed setting.

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 33.6MB gradient matrix in BF16, each GPU shares a 4096-dimensional vector (16KB), reducing communication volume by over 4,000× with negligible impact on importance estimation (Fig. 8(right)).

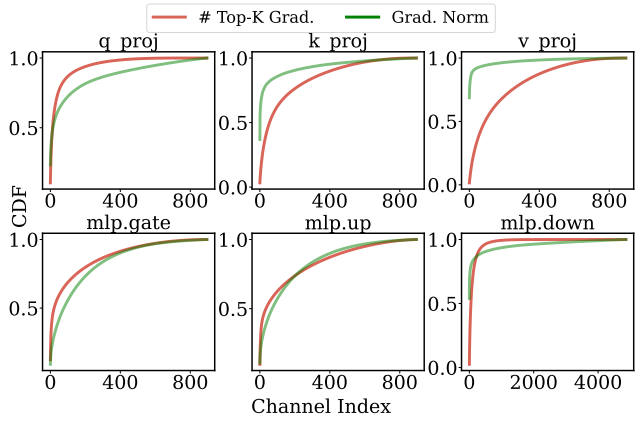


Figure 9. CDF of top-k elements and gradient norm. The channel index is sorted by the number of top-k elements.

Spatial Locality. We empirically validate the effectiveness of the approximation. Fig. 9 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 remain stable over time. Fig. 6b shows the retention rate—the fraction of previously selected channels that continue to contain top-k gradients across 100 steps. With a 10% selection threshold, nearly all top-k gradients are retained across iterations. Even with 3%-5% thresholds, retention remains high. This temporal stability suggests that it is effective to cache and reuse selected channel indices, rather than recomputing them at every step. In multi-GPU settings, this reduces the frequency of cross-device coordination, further amortizing selection cost and enabling scalable, importance-aware training.

3.4 Convergence Analysis

We now show that ZenFlow's asynchronous offloading design does not affect the convergence property of existing optimizers. We prove that ZenFlow achieves a 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 [11, 51].

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.

$$\theta_{t+1}^{(g)} = \theta_t^{(g)} - \alpha_t \nabla_{\theta^{(g)}} L(\theta_t)$$

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

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 (unbiased gradients, bounded variance, and Lipschitz-smooth) [11, 22, 24, 54], and letting ρ denote the fraction of gradient-energy at the CPU side, we have the following:

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

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.

Warm-up 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 $(0 < \beta < 1)$ [14, 19, 28].

ZenFlow exploits this observation by applying synchronous updates (i.e., no staleness) during the initial τ warm-up 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, when finetuning Qwen2.5-0.5B on Alpaca52K for three epoches with $T = 150{,}000$, $\tau = 7{,}500$ (5% warm-up), $S = 4$, $\rho = 0.1$, and $\beta = 0.6$ (typically ranging from 0.4 to 0.6 [19, 28]), the penalty is reduced from 0.18 to 0.12.

ZenFlow incurs only 0.12× penalty from the ideal SGD rate, which can be further reduced with modern optimizers like Adam/AdamW that mitigate gradient staleness via momentum and adaptive learning rates [21, 25, 26, 38].

With less than 0.12x penalty, ZenFlow can achieve 5× end-to-end speedup with 2× less I/O traffic and near-zero stall. We show the end-to-end speedup and detailed breakdown of performance gain in §5.

4 Implementation

We have implemented ZenFlow with approximately 11K lines of Python code. ZenFlow integrates seamlessly into DeepSpeed [35] without requiring any change to user training code. Our design extends DeepSpeed's ZeRO-Offload [37] and ZeRO-Infinity [34] 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, where a gradient tensor may be flattened and split on multiple GPUs, which means a single tensor of one input channel may be scattered on two device. 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 on devices. Additionally, we reorganize gradient storage from row-major to column-major layout by customizing PyTorch's [31] `flatten` operations to improve 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.

5 Evaluation

5.1 Experimental Setup

Testbed. The overall experimental environment is summarized in Table 2. 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.

Table 2. Experimental environments.

		A100 Testbed	H100 Testbed
HW	GPU	NVIDIA A100 (80GB) ×4	NVIDIA H100 (80GB) ×4
	CPU	AMD EPYC 7742 64C 128T (SMT Enabled)	Intel Xeon Platinum 8462Y 64C 128T (SMT Enabled)
	Memory	32×32GB DDR4-3200	32×64GB DDR5-4800
	PCIe	PCIe 4.0 × 16	PCIe 5.0 × 16
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}	

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 [46] following the standard setup in prior work [17], 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 [48], 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 [27] with a weight decay of 0.00 and apply a cosine learning rate schedule with 5% warmup. These settings are aligned with prior studies [8, 17, 52] to ensure fair, consistent comparison.

Baselines. We compare ZenFlow (ZF) against state-of-the-art offloading solutions, including **ZeRO-Offload** (ZO) [37] and **ZeRO-Infinity** [34]. We configure ZeRO-Offload with ZeRO Stage 2 [33] to maximize training throughput. For ZeRO-Infinity, we use default ZeRO Stage 3 [33, 34] and disable NVMe offloading to avoid potential performance degradation caused by frequent SSD access during training. We also implement the layer-wise scheduling technique from **StrongHold** (SH) [43] (not open-sourced) on top of DeepSpeed ZeRO-Offload to represent an optimized variant. 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 [17].

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 asynchronous offloading*, which selectively delays updates of

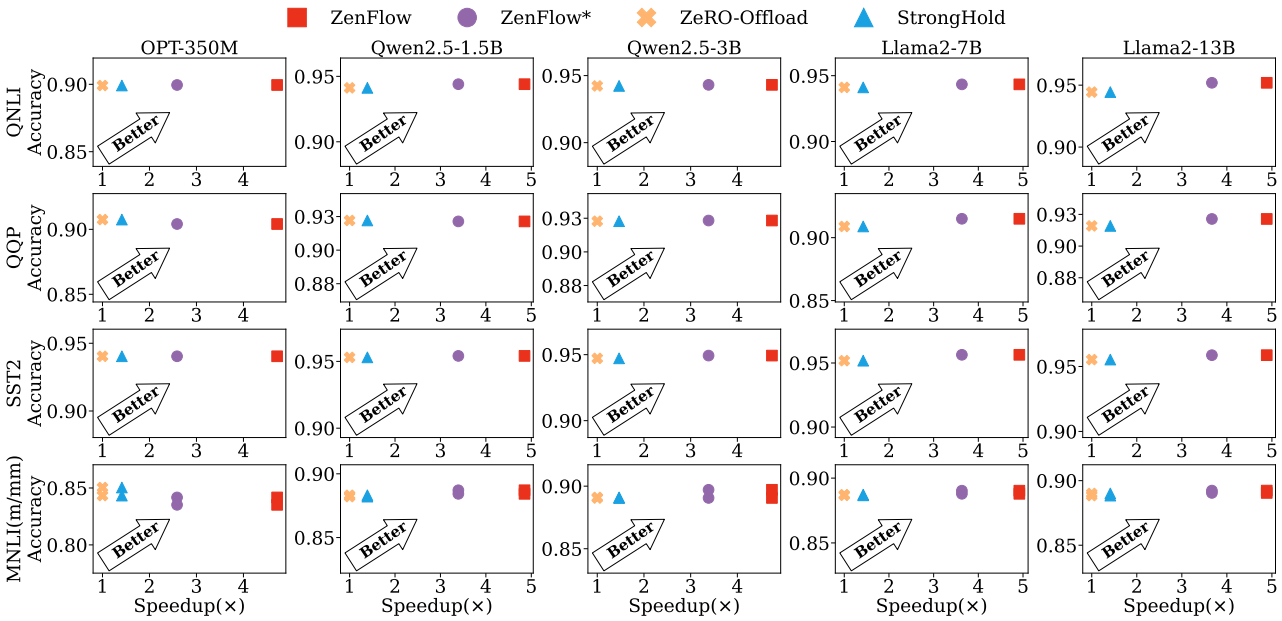


Figure 10. Accuracy vs. per-iteration speedup on GLUE tasks for various models and baselines.

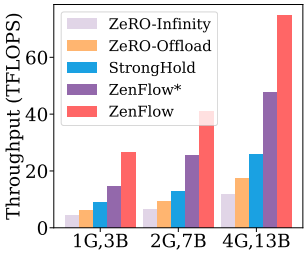


Figure 11. Throughput comparison across different model and GPU count configurations.

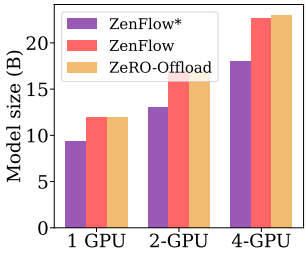


Figure 12. Maximum model size suppoted by each system as a function of GPU count.

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.

ZenFlow Hyperparameters. ZenFlow introduces two additional hyperparameters: the update interval S and the importance selection ratio `topk_ratio`. Unless otherwise specified, we set S=4, meaning less important parameters are updated once every 4 iterations (see §3.4). We provide a detailed analysis of these hyperparameter choices and their impact on accuracy and performance in §5.5.

5.2 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 [12].

ZeRO-Infinity consistently exhibits the lowest throughput due to communication overhead incurred by ZeRO-stage 3 [33]. 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× speedup over ZeRO-Offload and 6.3× 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 the 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,

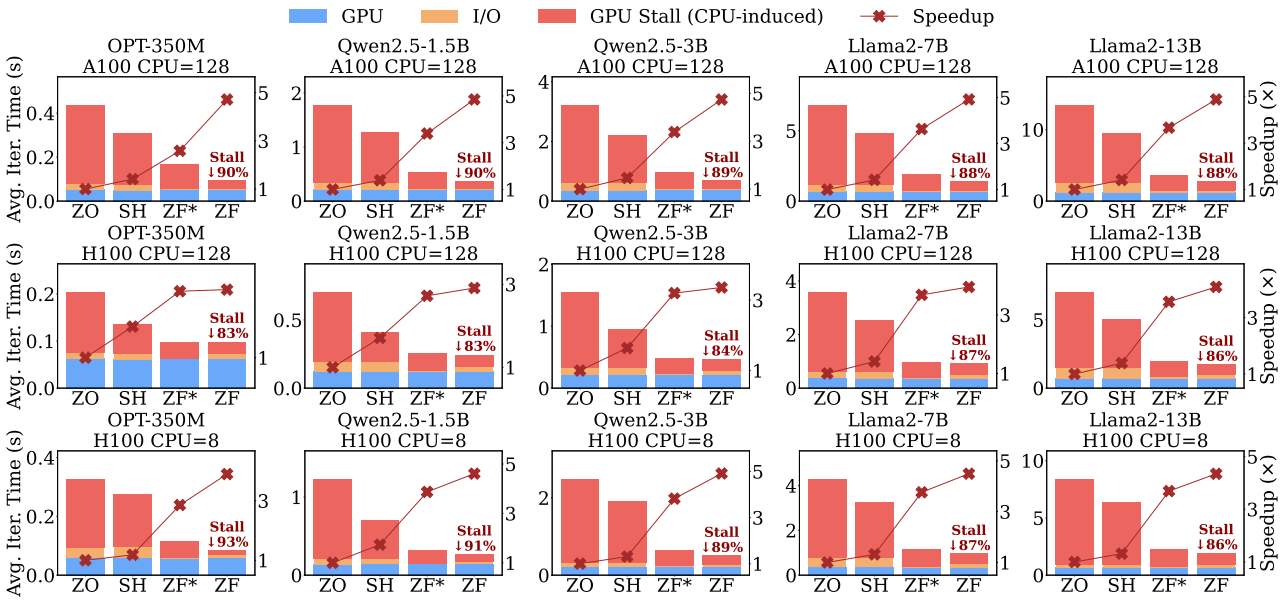


Figure 13. Training time breakdown and speedup across hardware configurations and models. Bars correspond to the left Y-axis. Lines (with markers) show relative speedup over ZeRO-Offload (ZO), referenced on the right Y-axis.

QNLI, QQP, and SST-2) across a range of model sizes and architectures. As shown in Fig. 10, 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 §5.5.

5.3 Time Breakdown and GPU Stall Analysis

We evaluate ZenFlow under three hardware configurations. The first setup reflects practical training environments with 4× A100 GPUs and full CPU parallelism. The second setup upgrades to 4× H100 GPUs with same CPU capacity. The third settings 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. 13, 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 2.9×-5× end-to-end speedup compared to ZeRO-Offload. For larger models (e.g., Llama2-7B and 13B), the benefits are more pronounced: the

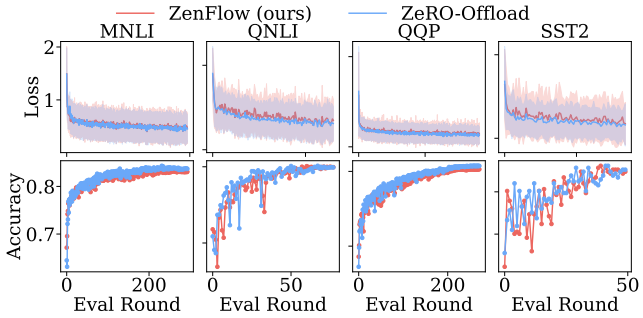


Figure 14. Convergence on GLUE with OPT-350M. ZenFlow matches ZeRO-Offload in both loss and accuracy across tasks.

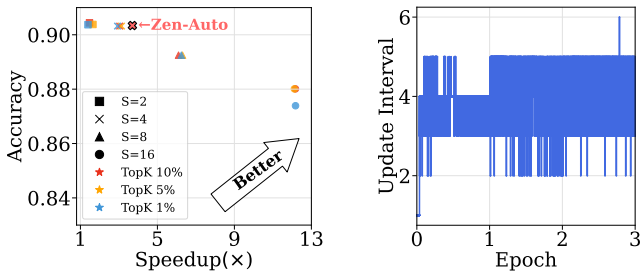
CPU-side overhead becomes a bottleneck due to heavy optimizer updates, even with highly parallelized, AVX-optimized `CPUAdam` optimizer. ZenFlow effectively mitigates this bottleneck, reducing the CPU:GPU compute time ratio from over $12:1$ (in baseline runs on 7B and 13B models) to $1:1$ or lower.

While ZenFlow introduces minor I/O overhead due to swapping *selective-optimizer* states out and in, these costs are effectively hidden by lightweight gradient selection and pipelined execution. The detailed breakdown of this overhead is presented in §5.6.

5.4 Convergence Validation

We evaluate the convergence behavior of ZenFlow compared to ZeRO-Offload on the GLUE benchmark using OPT-350M. As shown in Fig. 14, we report both the training loss (top row) and validation accuracy (bottom row) over evaluation rounds on four representative tasks: MNLI, QNLI, QQP, and

SST-2. 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). In terms of accuracy, ZenFlow matches or slightly outperforms ZeRO-Offload throughout training (e.g., QNLI), 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.



(a) Sensitivity analysis under varying top-k ratio (color-coded) and update interval S (shape-coded).

(b) Changes of update interval S over training with auto-tuning enabled.

Figure 15. Sensitivity analysis of key hyperparameters.

5.5 Sensitivity of Hyperparameters

We investigate the impact of hyperparameters in ZenFlow, specifically the top-k ratio and the update interval for CPU-side gradient handling. 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. 15(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. 15(a) we observe that ZenFlow is largely robust to variations in the top-k ratio in most cases, as ZenFlow preserves all gradients with bounded staleness. A higher ratio (e.g., 10%) yields slightly better accuracy (e.g., when S=2). However, at larger update intervals (e.g., S=16), smaller ratios may degrade accuracy due to increased staleness penalty (e.g., from 0.881 to 0.873 with 1% top-k). Since top-k selection does not significantly affect ZenFlow's performance efficiency, we opt for a high top-k ratio to ensure important channels consistently capture global top gradients. Empirically, we find that a 10% ratio is a good balance between accuracy and efficiency (see §2.3).

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. 15(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.

5.6 Communication Overhead Breakdown

ZenFlow's lightweight gradient gathering dramatically reduces communication volume and incurs minimal runtime overhead. As shown in Fig. 16, it achieves over 6,000× 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.

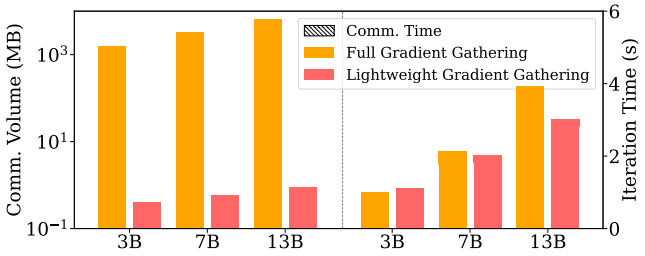


Figure 16. Overhead breakdown on gradient gathering.

6 Related Work

Tensor Offloading for Training. Prior works [4, 34, 37, 43] has explored offloading techniques to heterogeneous memory (e.g., CPU memory and NVMe SSD) to scale model training under GPU memory constraints. While these systems enable large-scale offloaded training, they largely *overlook the interplay between GPU and CPU* in the execution pipeline. Hybrid CPU-GPU training introduces significant GPU stalls due to CPU-side latency and inefficient PCIe transfers, especially during optimizer updates [29]. StrongHold [43] improves performance by exploiting the layer-wise model computation to overlap CPU and GPU computation. However, its effectiveness is limited by the performance gap between GPUs and CPUs, leaving CPU-induced stalls unresolved. These prior approaches treat all parameter updates uniformly, without considering the learning dynamics and hardware heterogeneity. In contrast, ZenFlow is both importance- and hardware-aware, decoupling important and non-important updates to minimize GPU stalls while maintaining training efficiency.

Gradient Sparsity and Compression. Prior studies have shown that gradients exhibit sparsity and can be effectively filtered using threshold-based selection [3, 13, 25, 42]. These works demonstrate that transmitting only the most significant gradients, e.g., top-k or above a threshold, is sufficient to preserve training quality while reducing communication or

offloading cost. In offloading settings, prioritizing important gradients has been extended to reduce I/O overhead. SmartInfinity [17] and LSP-Offload [8] propose improving I/O efficiency by applying lossy gradient compression—dropping small gradients or using learned projections. While effective in reducing I/O, such methods compromise gradient fidelity. These techniques are orthogonal and can be seamlessly integrated into ZenFlow's gradient offloading path, enabling further optimization without modifying its core scheduling and pipelining strategies.

Asynchronous Training. Many studies have explored asynchrony to accelerate distributed training [7, 23, 49, 50]. However, stale gradients may degrade the training performance and delay convergence [11, 24, 51, 54]. Extensive studies have been proposed to bound the staleness in training [7, 10, 24, 50]. In the context of offloaded training, ZenFlow is the first to address GPU stalls caused by CPU-side processing by leveraging bounded-asynchronous execution.

7 Conclusion

This paper presents ZenFlow, an importance-aware offloading framework that decouples GPU and CPU updates to eliminate GPU stalls and reduce I/O overhead in LLM fine-tuning. By updating important gradients in-place on the GPU and asynchronously accumulating the rest on the CPU, ZenFlow overlaps computation to minimize idle time. It leverages the spatial and temporal locality of gradients for scalable importance estimation without global synchronization. These techniques enable ZenFlow to significantly accelerate fine-tuning while improving GPU utilization and maintaining accuracy.

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] 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

[5] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*. Springer, 177–186.

[6] 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.

[7] 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.

[8] 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.

[9] 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

[10] 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.

[11] 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).

[12] DeepSpeed Team. 2025. DeepSpeed Flops Profiler. https://www.deepspeed.ai/tutorials/flops-profiler/. Accessed: 2025-05-16.

[13] 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.

[14] 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).

[15] 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

[16] 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

[17] 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

[18] 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.

[19] 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).

[20] 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

[21] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG] https://arxiv.org/abs/1412.6980

[22] Anastasiia 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.

[23] 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.

[24] 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).

[25] 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).

[26] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[27] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Bkg6RiCqY7

[28] 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).

[29] 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

[30] 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

[31] 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

[32] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[33] 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.

[34] 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

[35] 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.

[36] 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

[37] 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.

[38] Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive learning rates with sublinear memory cost. In *Proceedings of the 35th International Conference on Machine Learning*.

[39] 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

[40] 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.

[41] 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).

[42] Nikko Ström. 2015. Scalable distributed DNN training using commodity GPU cloud computing. (2015).

[43] 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.

[44] 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

[45] 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

[46] 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

[47] 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

[48] 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

[49] 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.

[50] 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.

[51] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. 2015. Staleness-aware async-sgd for distributed deep learning. *arXiv preprint arXiv:1511.05950* (2015).

[52] 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

[53] 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).

[54] 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.