10Cache: Heterogeneous Resource-Aware Tensor Caching and Migration for LLM Training

Sabiha Afroz
Virginia Tech, USA
sabihaafroz@vt.edu

Redwan Ibne Seraj Khan
Virginia Tech, USA
redwan@vt.edu

Hadeel Albahar
Kuwait University, Kuwait
hadeel.albahar@ku.edu.kw

Jingoo Han
Virginia Tech, USA
jingoo@vt.edu

Ali R. Butt
Virginia Tech, USA
butta@cs.vt.edu

ABSTRACT

Training large language models (LLMs) in the cloud faces growing memory bottlenecks due to the limited capacity and high cost of GPUs. While GPU memory offloading to CPU and NVMe has made large-scale training more feasible, existing approaches suffer from high tensor migration latency and suboptimal device memory utilization, ultimately increasing training time and cloud costs. To address these challenges, we present 10Cache, a resource-aware tensor caching and migration system that accelerates LLM training by intelligently coordinating memory usage across GPU, CPU, and NVMe tiers. 10Cache profiles tensor execution order to construct prefetch policies, allocates memory buffers in pinned memory based on tensor size distributions, and reuses memory buffers to minimize allocation overhead.

Designed for cloud-scale deployments, 10Cache improves memory efficiency and reduces reliance on high-end GPUs. Across diverse LLM workloads, it achieves up to 2× speedup in training time, improves GPU cache hit rate by up to 86.6×, and increases CPU/GPU memory utilization by up to 2.15× and 1.33×, respectively, compared to state-of-the-art offloading methods. These results demonstrate that 10Cache is a practical and scalable solution for optimizing LLM training throughput and resource efficiency in cloud environments.

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; *Planning and scheduling*; **Machine learning**; • **Software and its engineering** → *Development frameworks and environments*; • **Information systems** → *Hierarchical storage management*.

KEYWORDS

Deep Learning, LLM, Scheduling, Tensor Caching, Tensor Migration

ACM Reference Format:
Sabiha Afroz, Redwan Ibne Seraj Khan, Hadeel Albahar, Jingoo Han, and Ali R. Butt. 2025. 10Cache: Heterogeneous Resource-Aware Tensor Caching and Migration for LLM Training. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3772052.3772236

1 INTRODUCTION

Transformer-based large language models (LLMs) have become foundational in natural language processing and code generation due to their ability to capture complex context. As their accuracy improves with scale, LLMs continue to grow in size, reaching hundreds of billions or even trillions of parameters. Training such models, e.g., LLaMA 3 (70B) [61] or GPT-4 (1.76T) [11], demands massive compute and memory, often involving hundreds or thousands of GPUs. For instance, the 175B GPT model requires approximately 326 GB in FP16 format, which far exceeds the 80 GB capacity of a single NVIDIA H100 [8] GPU. These constraints have driven both industry and academia to seek training solutions that operate efficiently on cloud-scale infrastructure.

This explosive growth in model size has introduced new challenges for cloud systems. Public cloud platforms (e.g., AWS [18, 43], Azure [14, 56], GCP [52]) and private AI clusters face mounting pressure to maximize resource utilization and reduce the cost per training job. GPU memory constraints are especially acute: while GPU computational throughput continues to improve, memory capacity has not kept pace. For example, the NVIDIA H100 delivers 2× more FLOPS than the A100, yet offers only a marginal increase in memory size. This widening compute-memory imbalance makes GPU memory a primary bottleneck for scaling LLM workloads in the cloud, driving the need for efficient memory management and offloading strategies.

Many cloud users aim to reduce training costs by fine-tuning pre-trained LLMs on domain-specific tasks [23, 60]. Although fine-tuning requires fewer iterations and less data than full model training, it still consumes substantial GPU memory [27], often hitting the same memory wall [39] that limits full-scale training. Expanding to multiple GPUs is not always practical due to cost and resource constraints in cloud environments, making memory-efficient single-GPU fine-tuning a critical capability for workloads.

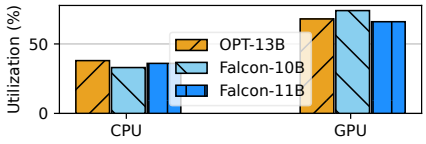


Figure 1: CPU and GPU memory utilization in ZeRO-Infinity.

To address memory bottlenecks in LLM training, especially in single GPU and cost-sensitive cloud environments, a range of techniques have been proposed, including mixed-precision arithmetic [20, 31, 44], data compression [16, 26], and memory offloading to CPU and NVMe storage [21, 22, 24, 25, 29, 30, 48, 50, 53]. Among these, memory offloading is a practical and widely adopted strategy to enable GPU memory oversubscription. However, offloading tensors to CPU or NVMe adds high data migration latency, increasing training time, and reducing hardware efficiency, which is detrimental in cloud-scale deployments.

Mitigating this overhead requires an efficient and latency-aware memory migration strategy. A well-optimized offloading mechanism must not only expand usable memory capacity but also preserve GPU throughput by minimizing data transfer times. Our key observation is that existing solutions fail to fully utilize the available system memory hierarchy, resulting in underutilization of both CPU and GPU during training. Thus, improving memory migration efficiency and resource utilization is essential for enabling fast, scalable, and cost-efficient LLM training in the cloud.

To better understand these challenges, we perform a motivational study. We train, by fine-tuning, three models (OPT-13B, Falcon-10B, and Falcon-11B) for one epoch using DeepSpeed ZeRO-Infinity [53] and observe their CPU and GPU memory utilization. ZeRO-Infinity offloads memory by partitioning model parameters, gradients, and optimizer states across CPU and NVMe, allowing larger models to fit within limited GPU memory. However, as shown in Fig. 1, ZeRO-Infinity achieves suboptimal CPU and GPU memory utilization, ranging between 38% and 74%, due to inefficient memory offload during training. This inefficiency in resource utilization directly impacts training performance, leaving significant room for improvement. These findings underscore the need for an offloading strategy that maximizes the usage of available CPU and GPU computational power while minimizing tensor migration latency to enhance training efficiency.

Recognizing these challenges, we introduce 10Cache, a lightweight and resource-aware tensor caching and migration framework that improves memory efficiency and training throughput in multi-tier memory systems. 10Cache targets three tiers of memory: GPU, CPU, and NVMe storage. It leverages a lightweight profiling phase to analyze tensor execution order, usage frequency, and size distribution. Based on this analysis, it constructs a prefetch table to proactively stage tensors in the fastest available memory (Table 1) before they are needed. To reduce the overhead of frequent allocations, 10Cache pre-allocates pinned memory buffers and reuses them across training iterations, eliminating repeated allocation costs while improving data transfer performance via direct memory access.

By intelligently aligning tensor placement with access patterns, 10Cache substantially reduces swap-in latency, boosts cache hit rates, and increases GPU and CPU memory utilization during training. These improvements allow large LLMs to be trained more efficiently using fewer or lower-cost GPUs by effectively leveraging underutilized CPU and NVMe memory tiers, making 10Cache particularly valuable for cost-sensitive and resource-constrained cloud environments, where maximizing hardware efficiency is critical. In practical fine-tuning scenarios, 10Cache improves training throughput while minimizing reliance on expensive multi-GPU setups, ultimately reducing operational costs and power consumption [47]. These capabilities offer tangible economic and environmental benefits for cloud providers and AI infrastructure operators.

Overall, the major contributions of this paper are summarized as follows:

- We analyze tensor execution order in deep learning workloads and propose a prefetching technique specifically designed for LLM training, improving training efficiency.

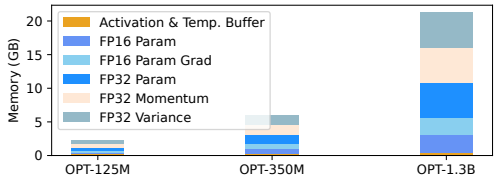


Figure 2: LLM model states memory breakdown.

- We introduce a dynamic hierarchical tensor allocation mechanism that distributes tensors across GPU, CPU, and NVMe memory, increasing the GPU cache hit rate.
- We design a novel resource-aware pre-allocated cache buffer for both CPU and GPU, reducing memory allocation overhead and enabling efficient memory management during tensor migration.
- We integrate 10Cache into the widely-used DeepSpeed framework [4] and compare it against eight state-of-the-art baseline approaches. Our evaluations show that 10Cache reduces the number of tensors with wait time below 0.03 ms by up to 1.92×, increases the GPU cache hit rate by up to 86.6×, CPU memory utilization by up to 2.15×, GPU memory utilization by up to 1.33×, and thus reduces training time by up to 2× compared to state-of-the-art methods.

Our results demonstrate that 10Cache is a practical, deployable system for improving LLM training efficiency in heterogeneous, memory-constrained environments, offering immediate impact for cloud-scale training platforms. 10Cache is publicly available at https://github.com/Sabiha1225/10cache.git

Table 1: Data transfer bandwidths across system components.

Transfer Type	Bandwidth	Transfer Type	Bandwidth
CPU-GPU	10.36 GB/s	CPU-NVMe Write	0.73 GB/s
GPU-CPU	9.51 GB/s	CPU-NVMe Read	2.36 GB/s

2 BACKGROUND AND MOTIVATION

2.1 Memory Demands in LLM Training

Training a DNN consists of three key steps: (1) forward pass, (2) backward pass, and (3) parameter update. In LLMs, these steps demand substantial memory, primarily due to model states, which include parameters, gradients, and optimizer states (e.g., momentum and variance in the Adam optimizer [37]) required for mixed-precision training (FP16/32) [44]. The remaining memory is consumed by activations and temporary buffer [49]. Mixed-precision training [44] with NVIDIA GPUs improves tensor core utilization [2, 49] by running forward and backward passes in FP16, storing parameters and activations in FP16 format. However, during the parameter update step with the Adam optimizer, large models require extra memory for FP32 copies of parameters, momentum, variance, and gradients. Specifically, for a model with N parameters, FP16 copies of parameters and gradients require $2N$ bytes each, while FP32 copies of parameters, momentum, and variance each require $4N$ bytes [49, 53]. Fig. 2 shows the memory breakdown for OPT-125M, OPT-350M, and OPT-1.3B, revealing that model states consume significantly more memory than activations and buffers.

Many previous works [29, 48] have addressed the GPU memory wall by offloading optimizer states to CPU memory and harnessing CPU computation for parameter updates when training large

Transformer-based models. However, these approaches have limitations. ZeRO-Offload [29] stores all model parameters in GPU, making GPU memory the limiting factor for training large models. In contrast, L2L [48] keeps only the current execution layer in GPU, leading to poor GPU memory utilization. For billion and trillion parameter models [11, 61], neither GPU nor CPU memory alone is sufficient. To address this, some works [36, 53] use NVMe storage offloading. ZeRO-Infinity enables offloading of both parameters and optimizer states to CPU and NVMe. However, existing offloading strategies often fail to utilize GPU memory efficiently when running large models, resulting in suboptimal memory usage. An effective solution should dynamically load model parameters to maximize GPU memory utilization without making the GPU a performance bottleneck. To this end, if the GPU retains only the tensors immediately needed for computation, up to its memory capacity, it can ensure timely access to minimize offloading-induced delays.

2.2 Impact of Pinned vs. Pageable Memory on CPU–GPU Transfer Efficiency

GPU memory offloading involves several data transfers between CPU and GPU, affecting LLM training time. CPU memory allocation uses two types: pageable and pinned memory. In pageable memory, the GPU cannot access data directly; the CUDA driver first creates a temporary pinned (page-locked) array [1], copies data from pageable memory to it, and then transfers it to GPU. Pinned memory serves as a staging area for transfers from GPU to CPU.

Our experimental results (Table 2) show that transferring data from CPU-pinned memory to GPU takes less than half the time compared to pageable CPU memory. This observation motivates the use of pinned cache memory in our proposed 10Cache system to reduce data transfer time during LLM training. Although DeepSpeed [4] uses pinned memory to store tensors, it keeps them in the same memory space throughout training. In contrast, 10Cache takes advantage of faster data transfers to and from the GPU by reusing pre-allocated pinned memory. Although pinned memory takes longer to allocate than pageable memory, 10Cache performs this allocation offline, so it does not add any overhead during model training. Additionally, frequent memory allocations in offloaded training increase training time, whereas reusing pre-allocated memory offers a more efficient solution. 10Cache leverages these insights for effective memory management.

Table 2: Data transfer time and bandwidth comparison: Pageable vs. Pinned memory.

Type	CPU-GPU (ms)	CPU-GPU (GB/s)	GPU-CPU (ms)	GPU-CPU (GB/s)	Data Type	Size (MB)
Pageable	1.65	10.16	1.68	10.00	FP32	16
Pinned	0.68	24.74	0.65	25.91	FP32	16
Pageable	0.78	10.69	0.89	9.48	FP16	8
Pinned	0.34	24.44	0.33	25.70	FP16	8

2.3 Tensor Behavior Analysis

A better understanding of tensor behavior in LLM training can help optimize GPU memory offloading and faster training. To study tensor execution patterns, active-inactive periods, and usage frequency, we generated the OPT-125M model trace using PyTorch's FX graph [5, 9]. The analysis reveals a tensor's life cycle: it is created during a PyTorch operation, used once or multiple times, and garbage collected when no longer needed.

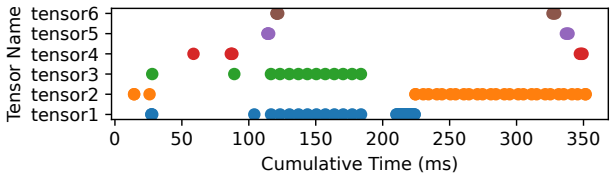


Figure 3: Tensors timeline for the OPT-125M model. Each dot marks a unique PyTorch operation at which the corresponding tensor becomes active.

From the OPT-125M FX graphs, we computed tensors' active and inactive times, usage frequency, and observed tensor size variations across layers. Fig. 3 shows the six most frequently used tensors and their timelines. Although created in different layers, these tensors are reused across multiple PyTorch operations within a single training iteration. The x-axis shows the cumulative execution time (ms), and the y-axis lists tensor names. Fig. 3 shows that "tensor1" is used at different times in multiple operations. This repeated usage pattern underscores the importance of tensor caching.

Active time indicates when a tensor participates in a PyTorch operation, while inactive time denotes periods of idleness. For example, in Fig. 3, "tensor2" becomes active at 15 ms (1st torch operation), remains active for 10 ms (till 2nd torch operation), then stays idle for 200 ms before becoming active again at 225 ms (3rd torch operation). During the 200 ms idle window, the tensor can be offloaded to CPU or NVMe memory and fetched back to GPU memory when needed, improving memory efficiency.

According to the FX graphs, about 90% of the kernels in the OPT-125M, OPT-1.3B, and OPT-2.7B models complete within 0.10 ms, 0.11 ms, and 0.12 ms, respectively, while about 50% finish within 0.09 ms. This motivates our choice of a 0.03 ms threshold for tensor wait time analysis (§ 4.2.2, § 4.2.5), as it remains well below typical kernel durations, minimizing the impact of swap-in latency on GPU throughput. To study system behavior across a range of tolerances, we also vary the threshold values (e.g., 0.01 ms, 0.1 ms) (§ 4.2.2).

In this work, we focus on LLM training workloads, which generally operate with a static execution graph and exhibit predictable repeated tensor operation patterns during training [21, 25]. Our FX-graph analysis further confirms this regularity. 10Cache is designed to exploit these repeated patterns to allow efficient tensor caching and smart prefetching and eviction. As a result, models with dynamic execution graphs, where tensor behavior is irregular and harder to predict, are beyond the scope of this study.

Training an LLM involves forward pass, backward pass, and parameter updates, requiring efficient memory management across GPU, CPU, and NVMe. While ZeRO-Infinity [53] stores larger tensors in CPU or NVMe memory, 10Cache keeps them in GPU memory if they are immediately required. G10 [21] considers the available bandwidth of flash storage and host memory when offloading

tensors. Ideally, larger tensors should reside in GPU memory when needed to ensure immediate availability and avoid delays.

3 10CACHE DESIGN

10Cache is an efficient training framework designed to scale large models on a single GPU, using heterogeneous memory tiers, including CPU and NVMe storage, to overcome GPU memory limits and improve training throughput over baselines [53, 59]. Its fine-grained, resource-aware tensor placement and dynamic prefetch-eviction strategies ensure optimal GPU memory usage. 10Cache seamlessly integrates into existing workflows, making it both powerful and user-friendly. This section outlines the 10Cache design.

3.1 System Overview

10Cache consists of four key components: the tensor characteristic analyzer, cache allocator, tensor allocator, and prefetch-eviction scheduler. Fig. 4 shows the workflow of the first three components. The tensor characteristic analyzer extracts tensor execution order and size from the model to understand tensor behavior. Using this information, the cache allocator assigns cache buffers across GPU and CPU memory to maximize memory efficiency, while the tensor allocator places tensors to minimize unnecessary offloading. During training, the scheduler asynchronously prefetches and evicts tensors, overlapping data transfers with GPU computation (Fig. 7, Fig. 8). The following sections detail each component.

3.2 Tensor Characteristic Analyzer

To efficiently utilize limited and costly GPU and CPU memory, 10Cache applies intelligent tensor migration policies that exploit heterogeneous memory while minimizing offloading overhead. This requires understanding tensor behavior and memory demands. The tensor characteristic analyzer performs a dry run of the model to capture execution patterns and tensor sizes. Based on this analysis, 10Cache builds a prefetch table and characterizes tensor sizes.

3.2.1 Prefetch Table Creation. When utilizing multi-tiered storage to offload tensors for large-model training, the tensor prefetch-eviction scheduler must anticipate when a tensor will become active in a GPU kernel operation. Without this foresight, delayed tensor retrieval can lead to GPU stalls and prolonged training times. To enable efficient prefetching and eviction, 10Cache builds a prefetch table that records each tensor's execution order, activation time, current location, and final location. The current location indicates where the tensor resides during training (CPU, GPU, or NVMe), while the final location refers to the optimal memory tier assigned by the tensor allocator before training begins. After each iteration, tensors are restored to their final locations to maintain optimal placement. PyTorch [10] allows registering `pre_hook` and `post_hook` functions that run before and after a layer executes during forward and backward passes. 10Cache employs these hooks to track tensor execution patterns, order and activation times. From this analysis, it builds a prefetch table which is later used for tensor placement (§ 3.3.2) in multi-tiered memory and for prefetch-eviction of tensors during training (§ 3.4). 10Cache utilizes the observation that the tensor execution order remains consistent and repetitive throughout the DNN training [25] (§ 2.3). Therefore, capturing and using this execution order for tensor prefetching and eviction during training becomes both effective and desirable.

3.2.2 Tensor Size Characterization. To manage memory efficiently and reduce the frequent allocations overhead during tensor offloading, 10Cache pre-allocates dedicated memory buffers for tensor caching. A key challenge in pre-allocation is the variation in tensor sizes within a model. A uniform memory allocation would lead to internal and external fragmentation [19, 65], wasting valuable GPU and CPU memory critical for LLM training. To address this, the tensor characteristic analyzer performs a lightweight dry run to profile and categorize tensor sizes, incurring minimal overhead compared to total training time (§ 4.2.9). Based on this, 10Cache optimally allocates cache memory between GPU and CPU, reducing fragmentation and maximizing memory efficiency. 10Cache organizes buffers by tensor sizes to ensure seamless tensor caching and retrieval during training (§ 3.3.1).

3.3 Cache Allocator

As analyzed in § 2.2, tensor swap-in and swap-out memory operations are costly. To reduce this overhead, 10Cache adopts two key strategies. First, it performs memory allocation once and reuses the same memory for all subsequent swap-in and swap-out operations. This eliminates the time spent on repeated allocations during training, leading to improved training time (§ 4.2.1). An additional advantage of reusing memory is that it reduces fragmentation, which often results from frequent allocations during training. This design choice not only speeds up training, but also ensures efficient memory utilization. Second, 10Cache strategically allocates memory in pinned CPU space to speed up CPU–GPU data transfers. Although pinned memory allocation takes longer than pageable memory, 10Cache performs this allocation once before training begins, so it does not impact the overall training time.

3.3.1 Buffered Memory Allocation. A key challenge lies in allocating memory that can be reused without creating internal or external fragmentation. As observed in § 2.3, tensor sizes vary across different model layers. While one might consider allocating all buffers to match the size of the largest tensor in both CPU and GPU, this naive approach leads to internal memory fragmentation. For example, when storing a 512-byte tensor in a 1024-byte buffer (sized for the largest tensor), half of the buffer space goes unused. A more efficient approach analyzes the specific tensor size distribution of each model, allowing optimized buffer allocation that minimizes memory waste. The pre-allocation process involves three steps: 1) calculating the tensor size distribution, 2) determining the required buffer count per size, and 3) allocating memory accordingly.

Step 1: Tensor Size Distribution Calculation. This process takes as input a dictionary TC, where each entry (s_i, c_i) represents the tensor size (s_i) and its corresponding count (c_i). It then computes the total memory requirement in bytes for each unique tensor size (Alg. 1, lines 2–5). Subsequently, it calculates the memory requirement ratio of each tensor size relative to the total memory requirement for all tensors (Alg. 1, lines 6–7).

Step 2: Buffer Count by Tensor Size in Cache Memory. This process computes the number of buffers for each tensor size based on available GPU and CPU memory. It takes as input the tensor

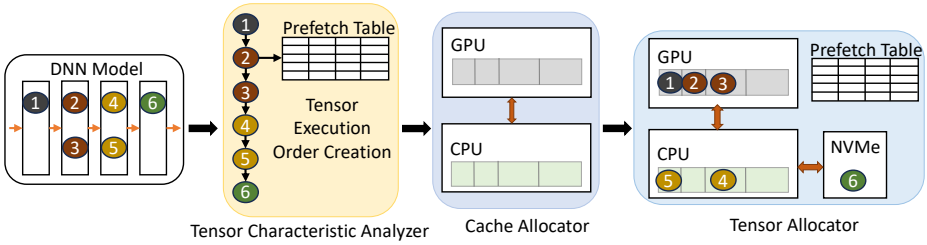


Figure 4: 10Cache's three components' end-to-end flow from profiling a DNN model to tensor allocation across CPU, GPU and NVMe memory. Tensors from the same layer share the same color, while different layers use distinct colors.

Algorithm 1: Tensor Size Distribution Calculator

Data: $TC = \{(s_1, c_1), (s_2, c_2), \dots\}$; // TC = a dict for tensor count (c_i) for each size (s_i)
Result: $TSD = \{\}$; // TSD = a dict holding the distribution for each size

1 total_size ← 0;
2 **foreach** $(s_i, c_i) \in TC$ **do**
3 $sc \leftarrow s_i * c_i$;
4 total_size ← total_size + sc;
5 $TSD[s_i] \leftarrow sc$;
6 **foreach** $s_i \in TSD$ **do**
7 $TSD[s_i] \leftarrow TSD[s_i]/total_size$;

Algorithm 2: Buffer Counter

Data: $TC = \{(s_1, c_1), (s_2, c_2), \dots\}$, TSD ; // TC = a dict for tensor count (c_i) for each size (s_i), TSD = a dict holding the distribution for each size
Result: GPU_BUFFER_COUNT ← {}, CPU_BUFFER_COUNT ← {} ; // two dict for holding buffer count for each size

1 gpu_avl_mem ← memory_profiler.get_gpu_free_mem();
2 cpu_avl_mem ← memory_profiler.get_cpu_free_mem();
3 **foreach** $(s_i, c_i) \in TC$ **do**
4 GPU_BUFFER_COUNT$[s_i]$ ← min($(TSD[s_i] *$ gpu_avl_mem$)/s_i, c_i$);
5 CPU_BUFFER_COUNT$[s_i]$ ← min($(TSD[s_i] *$ cpu_avl_mem$)/s_i, c_i$ − GPU_BUFFER_COUNT$[s_i]$);

size distribution from Step 1 and uses two memory profiler APIs to obtain the available GPU and CPU memory (Alg. 2, lines 1-2). Using these data, the buffer counter computes the required buffers per tensor size, first for FP16 parameters on GPU (Alg. 2, line 4), then for FP16 tensors on CPU (Alg. 2, line 5), and finally for FP32 optimizer states.

Step 3: Memory Allocation. This step utilizes the buffer counts computed in Step 2 (GPU_BUFFER_COUNT, CPU_BUFFER_COUNT) to determine the total memory required for each tensor type. To reduce fragmentation, 10Cache allocates a contiguous region in pinned memory and partitions it into fixed-size chunks based on buffer sizes. A free buffer list tracks available buffers for reusability.

From Fig. 5, the table outlines the number of buffers required for each tensor size. Below it, the diagram represents a contiguous memory block. The allocation process begins by assigning the first 512-byte chunk as 'buffer0', which is then added to the free buffer list. This process continues sequentially, allocating memory chunks and registering them in the free buffer list according to their sizes. When the system requires a 512-byte buffer, it first retrieves 'buffer0' from the free list, ensuring efficient memory utilization while reducing allocation overhead.

3.3.2 Tensor Allocator. Our proposed 10Cache efficiently adapts to available memory resources. When GPU and CPU memory can accommodate model parameters and optimizer states, it confines storage to these high-speed memory tiers. However, when the model size exceeds the combined capacity of GPU and CPU memory, 10Cache dynamically distributes parameters and optimizer states across GPU, CPU, and NVMe storage. This strategic placement

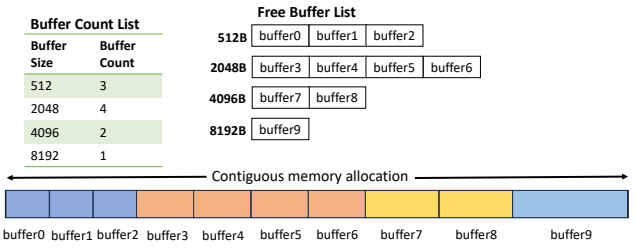


Figure 5: Cache buffer allocation strategy (A contiguous memory block is partitioned into fixed-size buffers for different tensor sizes. Buffers are registered in a free list by size.)

enhances training efficiency and optimizes resource utilization, ensuring minimal offloading overhead and improved performance.

An Illustrative Example. Fig. 6 illustrates the execution sequence of FP16 parameters during the forward and backward passes in one training iteration. The forward pass begins with layer 1, so tensors 1 and 2 from this layer are accessed immediately. If these tensors are large, ZeRO-Infinity [53] stores them in CPU or NVMe memory, which introduces migration latency, as they must be brought back to GPU memory for computation. In contrast, 10Cache leverages knowledge of tensor execution order to keep these immediately required tensors in GPU memory, significantly reducing migration latency. To optimize tensor placement, 10Cache's tensor allocator strategically distributes tensors across the tiered memory. It uses the prefetch table to identify tensors needed soon and prioritizes placing them in GPU cache. An active

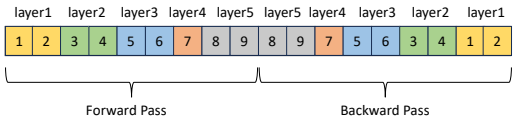


Figure 6: A single training iteration for the model.

tensor window tracks tensors in GPU memory, while additional tensors are stored in CPU cache, with any remaining placed in NVMe if CPU memory is full.

In a scenario where CPU and GPU cache memory can accommodate the entire model, the 10Cache tensor allocator ensures that tensors 1, 2, 3, and 4 remain in GPU memory, while tensors 5, 6, 7, 8, and 9 are stored in CPU. As the scheduler loads tensors 6, 7, 8, and 9 into GPU for execution, it retains them for immediate reuse during backpropagation. This approach significantly increases the GPU cache hit rate (§ 4.2.3 and § 4.2.7) and reduces the frequency of tensor offloading. The prefetch table maintains an up-to-date record of tensor locations across heterogeneous memory, enabling the scheduler to make informed prefetching and eviction decisions.

3.4 Training

During model training, the 10Cache scheduler efficiently manages tensor prefetching and eviction by overlapping GPU computation with CPU-GPU communication, thus minimizing data transfer overhead. For LLM training, the CPU optimizer [29] performs parameter updates on the CPU, improving memory efficiency, since optimizer states consume substantial memory (§ 2.1) and GPU memory is both limited and expensive. Based on model size, 10Cache dynamically employs either a CPU-GPU or an extended CPU-GPU-NVMe offloading strategy to scale large model training. To ensure robustness under runtime noise (e.g., OS jitter, multi-tenant contention) in production environments, 10Cache avoids reliance on wall-clock timing. Instead, it uses lightweight PyTorch pre-/post-hooks (§ 3.2.1) that trigger prefetching based on actual layer execution, ensuring the scheduler remains synchronized even when execution drifts from the profiled timing.

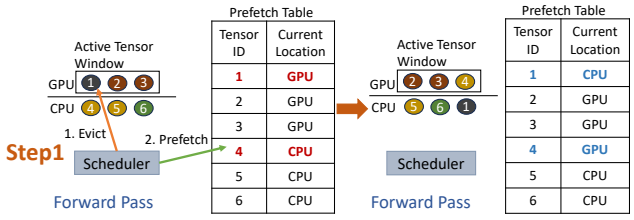


Figure 7: CPU-GPU Offloading: Prefetches tensor 4 and evicts tensor 1 during the forward pass based on execution order.

3.4.1 CPU-GPU Offloading. If the aggregate memory of the CPU and GPU is sufficient to store the model parameters and optimizer states, 10Cache prevents storing any tensors in slower NVMe memory. In this scenario, the CPU cache stores all the optimizer tensors, while parameter tensors are distributed across the GPU and CPU cache. Fig. 7 demonstrates an example of tensor prefetching and eviction for CPU-GPU offloading. For example, the DNN model has parameter tensors 1, 2, 3, 4, 5, 6. 10Cache's tensor allocator places tensors 1, 2, and 3 in the GPU memory and tensors 4, 5, and 6 in CPU memory (Step1). During forward pass, once tensor 1 finishes execution, the scheduler evicts (1) it to CPU (Alg. 3, lines 1-4) and asynchronously prefetches (2) tensor 4, which is not in active tensor window, ahead of time (Alg. 3, lines 5-13). Therefore, when tensor 4 is needed for kernel execution, it will already be in the GPU. After eviction and prefetching, the GPU stores tensors 2, 3, 4, and the CPU stores tensors 1, 5, 6. Similarly, the scheduler evicts tensors 2 and 3 in CPU and prefetches tensors 5 and 6 in GPU during forward pass. When the activation window contains tensors 4, 5 and 6 in GPU, the scheduler halts any eviction or prefetching, as these tensors are essential for immediate backward pass. Offloading them would require reloading, incurring unnecessary overhead. By avoiding such redundant transfers, 10Cache's scheduling policy improves both training time (§ 4.2.1) and cache hit rate (§ 4.2.3).

Algorithm 3: PrefetchTensor

```
Input: evicted_tensor_list
1  foreach evict_tensor_id in evicted_tensor_list do
2      if prefetch_tab_cur_row < len(prefetch_table) then
3          Remove evict_tensor_id from active_tensor_window
4          EvictTensor(evict_tensor_id)
5          while prefetch_table[prefetch_tab_cur_row].tensor_id
             ∈ active_tensor_window and prefetch_tab_cur_row
             < len(prefetch_table) do
6              prefetch_tab_cur_row++
7              release_param ← False
8          prefetch_tensor_id ←
             prefetch_table[prefetch_tab_cur_row].tensor_id
9          Add prefetch_tensor_id to active_tensor_window
10         Get a free GPU buffer ID for prefetch_tensor_id
11         if prefetch_table[prefetch_tab_cur_row].current_loc is
             'cpu' then
12             Get CPU buffer ID for prefetch_tensor_id
13             Async copy from CPU buffer to GPU buffer
14         else
15             if
                 prefetch_table[prefetch_tab_cur_row].current_loc
                 is 'nvme' then
16                 Async copy from NVMe to CPU temp buffer,
                     then to GPU buffer
17         prefetch_tab_cur_row++
```

3.4.2 CPU-GPU-NVMe Offloading. For large models, CPU and GPU memory alone cannot accommodate all model parameters and optimizer states. In such cases, 10Cache's tensor allocator strategically stores a portion of the FP16 parameters and optimizer states in NVMe to scale LLM training.

FP16 Parameter Scheduling Policy. Fig. 8 demonstrates prefetching and eviction in a CPU-GPU-NVMe setup. In Step1, during the forward pass, tensors 1, 2, and 3 complete execution and 10Cache offloads them to CPU cache. Tensors 4, 5, and 6 remain active in GPU memory, while tensor 7 is placed in NVMe by tensor allocator due

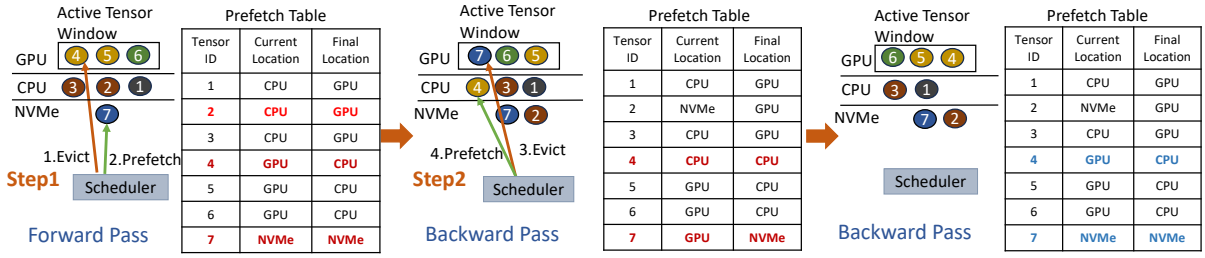


Figure 8: CPU-GPU-NVMe offloading: when memory is limited, 10Cache's scheduler coordinates tensors across all three tiers.

Algorithm 4: EvictTensor

Input: evict_tensor_id

```
Get final_loc, evict_tensor_size, and GPU buffer ID for evict_tensor_id
if final_loc is 'nvme' then
    Release GPU buffer for evict_tensor_id
    Update current location to 'nvme'
else
    if CPU has free buffer of size evict_tensor_size then
        Get a free CPU buffer ID
    else
        if CPU has occupied GPU-designated buffer of evict_tensor_size then
            Get its tensor ID and buffer ID
            Swap tensor to NVMe, update location to 'nvme'
            Release GPU-designated CPU buffer
        else
            Get any occupied CPU buffer ID and tensor ID of evict_tensor_size
            Swap tensor to NVMe, update location to 'nvme'
    if final_loc is 'gpu' then
        Mark buffer as GPU-designated in CPU cache
    Mark CPU buffer as occupied by evict_tensor_id
    Async copy from GPU to CPU buffer
    Release GPU buffer for evict_tensor_id
    Update current location to 'cpu'
```

to limited CPU and GPU memory. Once tensor 4 completes execution during the forward pass, the scheduler attempts to evict it to the CPU cache (1). As the CPU cache lacks free buffer space and tensor 2 (currently in the CPU) has the same size as tensor 4 while remaining inactive for a longer period, 10Cache evicts tensor 2 to NVMe storage to create space for tensor 4 as tensor 4 will be used earlier in the backward pass. Now, 10Cache's scheduler marks tensor 7 for prefetch (2). 10Cache first copies the tensor into a temporary buffer in CPU memory before asynchronously transferring it to the GPU buffer (Alg. 3, lines 14-16). This ensures uninterrupted GPU computation. NVMe always holds tensor 7 copy. When needed, this tensor is prefetched to GPU from NVMe. For eviction, only the GPU memory buffer is released since the NVMe copy is already available.

If GPU buffers for a tensor size are exhausted, 10Cache evicts a tensor not needed soon. If it already resides in NVMe, the GPU buffer is released (Alg. 4, lines 2-4). Otherwise, it checks for a free CPU buffer to offload the tensor and releases the GPU buffer (Alg. 4, lines 6-7, 16-21). If CPU buffer is unavailable, it evicts a GPU-designated CPU tensor to NVMe and reuses it (Alg. 4, lines 9-12, 16-21). A GPU-designated CPU tensor is one initially placed in GPU memory by the tensor allocator but later offloaded to CPU after its forward-pass completes. When the CPU buffer is full for this particular tensor size, 10Cache's scheduler selects a GPU-designated CPU tensor for eviction, as it will be needed later during backpropagation. The tensor allocator prioritizes placing tensors in GPU, then CPU, and finally NVMe storage. During backpropagation, tensors are used in reverse order, starting with those in NVMe, followed by CPU, and then GPU. 10Cache's scheduler is aware of this order and evicts tensors accordingly to minimize offloading overhead. If none is found, it evicts any occupied CPU buffer to NVMe and reuses the buffer (Alg. 4, lines 13-21). The tensor allocator sets each tensor's location in the prefetch table (§ 3.3.2).

After prefetching tensor 7 and evicting tensor 4, GPU stores tensors 5, 6 and 7, CPU holds tensors 1, 3 and 4 and NVMe stores tensors 2 and 7. 10Cache's scheduler now halts any eviction and prefetch operations as tensors 5, 6, 7 will be used immediately in backward pass. In Step2, once tensor 7 execution finishes during the backward pass, the scheduler evicts (3) it. As NVMe already holds a copy of tensor 7, 10Cache releases the GPU buffer of tensor 7 back to the free buffer list (Alg. 4, lines 2-4), allowing it to be reused later by another tensor of similar size. As tensor 4 needs to be processed during the backward pass, it is prefetched (4) from the CPU cache using a swap-in operation to the GPU cache (Alg. 3, lines 11-13). Tensor 4 has the same size as tensor 7 and hence reuses the free buffer of tensor 7 in the GPU cache which was evicted previously. After eviction and prefetching, GPU stores tensors 4, 5 and 6, CPU stores tensors 1 and 3 and NVMe stores tensors 2 and 7.

Optimizer States Scheduling Policy. For LLMs, optimizer states consume substantial memory (§ 2.1), often exceeding available CPU capacity. A conventional offloading approach like ZeRO-Infinity [53] stores all optimizer states in NVMe due to limited CPU memory, but this introduces inefficiencies. Assume that we have three FP32 parameter tensors: T1, T2, and T3. In ZeRO-Infinity, these tensors and their optimizer states reside in NVMe. During the optimizer step, the scheduler first loads T1 and its corresponding optimizer state into CPU memory. After updating T1, it moves the tensor back to NVMe and then fetches T2. This synchronous swap-in and swap-out process increases training time and leaves CPU memory underutilized.

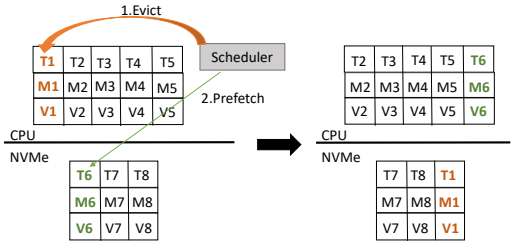


Figure 9: Optimizer states prefetching and eviction.

To overcome this inefficiency, 10Cache's tensor allocator dynamically distributes optimizer state tensors between CPU cache and NVMe based on available CPU memory capacity. As illustrated in Fig. 9, 10Cache stores optimizer states for tensors T1 through T5 in CPU, while T6, T7, and T8 are stored in NVMe. Once the parameter update for T1 is complete, the scheduler evicts it (1) and asynchronously prefetches T6 (2). This asynchronous prefetching overlaps data transfer time from NVMe to CPU with CPU computation. This strategy improves both training time (§ 4.2.4) and CPU memory utilization (§ 4.2.8).

4 EVALUATION

4.1 Experimental Setup

4.1.1 Implementation Details. 10Cache is built on top of DeepSpeed [4], a PyTorch-based [9] framework optimized for LLM training. It extends ZeRO-Infinity [53] with efficient parameter and optimizer states placement across heterogeneous memory, intelligent tensor migration scheduling, and advanced memory management to enhance training throughput. 10Cache integrates seamlessly with DeepSpeed, supporting any compatible LLM workflow for practical and efficient training. For profiling, it uses `nvidia-smi` to monitor GPU memory and `Psutil` to track CPU memory.

4.1.2 System Configurations. Table 3 summarizes the experimental setup. In all configurations, the Samsung NVMe connects to the CPU via PCIe 4.0 ×4, and the GPU connects to the CPU via PCIe 4.0 ×16. We use the NVIDIA L40S GPU as the default, unless stated otherwise. All experiments use PyTorch 2.3.0 and DeepSpeed 0.14.2.

Table 3: System configurations.

CPU	2x Intel Xeon Silver 4314 (16c, 32t)	AMD EPYC 7763 64-Core Processor
GPU	NVIDIA L40S (48GB), A40 (48GB)	A100 (40GB)
Memory	256GB, 200GB	256GB
Storage	2TB Samsung NVMe	3TB Samsung NVMe

4.1.3 Workloads and Datasets. We evaluate 10Cache using diverse LLMs and datasets summarized in Table 4. To emulate limited GPU memory scenarios, we use a default batch size of 16 and vary it in § 4.2.10 to analyze its effect on training time.

4.1.4 Baselines. We evaluate 10Cache against eight recent state-of-the-art GPU memory swapping approaches. We ensure a fair evaluation by using the default or recommended configurations

Table 4: LLM workloads and datasets.

Model	Source	Dataset
OPT-6.7B,OPT-13B [64]	Hugging Face	GLUE MRPC
Bloom-7B [54]	Hugging Face	GLUE COLA
Falcon-7B,Falcon-10B	Hugging Face	Wikitext
Falcon-11B [12], GPT2	Hugging Face	Wikitext

of baselines without extra prefetching or memory tuning beyond what these frameworks provide.

ZeRO-Offload [29] stores model parameters in GPU memory and optimizer states in CPU memory, relying on CPU computation for parameter updates. ZeRO-Infinity [53] extends offloading across CPU, GPU, and NVMe. L2L [48] reduces GPU memory pressure by keeping only the active layer in GPU. StrongHold [59] improves upon L2L by storing multiple layers to reduce offloading overhead. Megatron-LM [57] enables large transformer training through model parallelism. FlashNeuron [30] offloads intermediate tensors to NVMe SSD via direct GPU–SSD communication [6]. DeepUM [25] uses CUDA Unified Memory [3] with a correlation-based prefetcher for GPU memory oversubscription. G10 [21] unifies host, GPU, and flash memory into a single memory space, scheduling tensor migration based on available bandwidth of flash and host memory.

For CPU-GPU offloading, we compare 10Cache with ZeRO-Offload, ZeRO-Infinity, L2L, StrongHold, and Megatron-LM. For CPU-GPU-NVMe offloading, we evaluate it against ZeRO-Infinity, FlashNeuron, DeepUM, and G10. In CPU-GPU setups, 10Cache denotes the base version without prefetching or memory optimizations. 10Cache+P adds prefetching using a prefetch table for timely tensor loading and eviction, and 10Cache+P+M adds pre-allocated buffers to reduce allocation overhead. For CPU-GPU-NVMe setups, 10Cache+FP16 applies all optimizations for FP16 parameters, while 10Cache+FP16+Opt further includes prefetching and eviction of optimizer states.

4.2 Performance Analysis

4.2.1 Training Time Evaluation (CPU-GPU Offloading). Fig. 10 compares the training times of OPT-6.7B, Bloom-7B, and Falcon-7B, showing how 10Cache improves LLM training efficiency over baselines. In this experiment, we restrict GPU memory to 24 GB. ZeRO-Offload fails with GPU OOM error for 7B range models since it stores parameters in GPU memory and optimizer states in CPU memory (§ 4.1.4), requiring about 28 GB GPU memory for FP16 parameters and gradients, despite sufficient CPU RAM. ZeRO-Infinity supports parameter offloading to CPU memory, allowing training under this constraint. 10Cache+P+M reduces training time by about 2× compared with ZeRO-Infinity for all three models, while the base 10Cache performs worse due to the lack of prefetching and pre-allocated cache memory. L2L performs the worst, loading one layer at a time and offloading it after execution, causing high communication overhead and GPU stalls.

Fig. 11a compares the training times of 10Cache+P+M, StrongHold, ZeRO-Infinity, and Megatron-LM for the GPT-2 [11] 5.9B model on NVIDIA L40S and NVIDIA A100 GPUs. In both settings, 10Cache+P+M outperforms ZeRO-Infinity and StrongHold, achieving speedups of 2.60× and 2.44× on L40S, and 2.82× and 4.17×

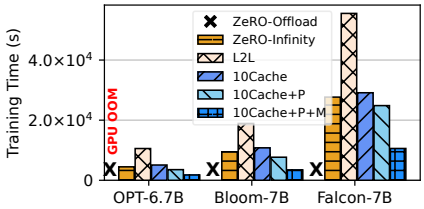
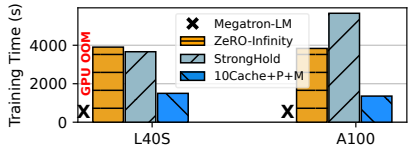
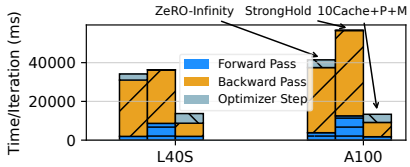


Figure 10: Training performance of 10Cache vs. baselines under CPU-GPU offloading.

on A100. To better understand this performance gain, Fig. 11b presents a breakdown of one iteration's execution time. The improved backward pass time of 10Cache+P+M is attributed to its strategic caching and prefetch-eviction mechanisms. StrongHold attains the fastest optimizer step but suffers higher forward and backward times due to frequent CPU-GPU layer transfers. Megatron-LM lacks offloading mechanism and fails to train the 5.9B model on a single GPU, resulting in a CUDA Out-Of-Memory error.



(a) Training efficiency comparison



(b) Per-iteration time breakdown

Figure 11: Training time comparison on L40S and A100 GPU setups (5.9B model).

We evaluate 10Cache on smaller models, GPT-2 [11] 1.7B and 4.0B, using an NVIDIA L40S GPU, and compare it with Megatron-LM, L2L, StrongHold, and ZeRO-Infinity in terms of training time (Fig. 12). 10Cache+P+M achieves speedups of 1.67× and 1.45× for the 1.7B model, and 1.81× and 1.58× for the 4.0B model, compared to ZeRO-Infinity and StrongHold, respectively. Megatron-LM encounters a GPU OOM error on the 4.0B model due to the lack of offloading, but performs well on the 1.7B model since all computations remain on the GPU.

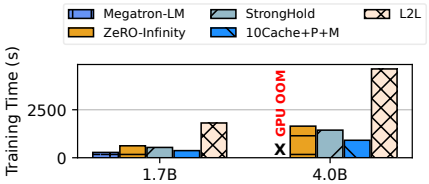


Figure 12: Training time comparison (1.7B and 4.0B models).

4.2.2 Wait Time Analysis (CPU-GPU Offloading). The wait time metric measures how well parameter transfers from CPU to GPU overlap with GPU computation. Fig. 13 compares this metric for 10Cache+P+M and ZeRO-Infinity, providing a fair comparison since both offload parameter tensors, unlike L2L [48] and StrongHold [59], which move model layers between CPU and GPU. Fig. 13 shows the percentage of tensors with wait times below 0.03 ms (§ 2.3), computed as ((tensors' count with wait time below 0.03 ms * 100) / total tensor count). With pre-allocated GPU cache, 10Cache loads tensors before execution for direct GPU access, while its prefetching further overlaps data transfers with computation more effectively than ZeRO-Infinity. Consequently, for OPT-6.7B, Bloom-7B, and Falcon-7B, 10Cache+P+M achieves 1.36×, 1.19×, and 1.92× more tensors with wait times below 0.03 ms than the baseline.

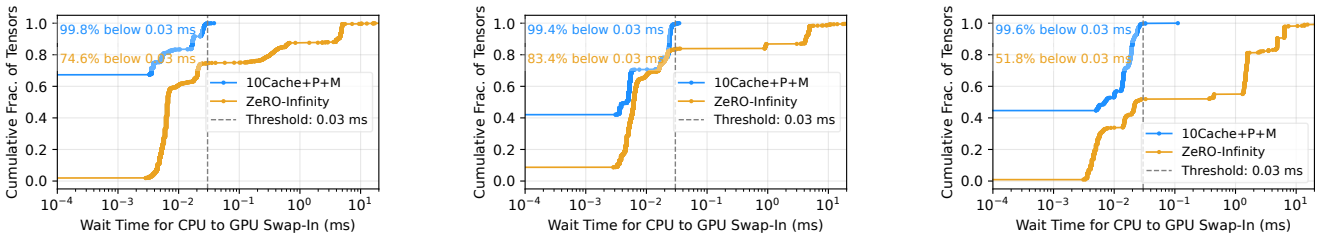
Fig. 14 shows the proportion of tensors with CPU-to-GPU transfer wait times below 0.01 ms and 0.1 ms. With a 0.01 ms threshold, about 83% of tensors meet the target using 10Cache, and at 0.1 ms, nearly all tensors fall below the threshold, highlighting 10Cache's ability to minimize swap-in delays across varying tolerances.

4.2.3 GPU Cache Hit Rate Analysis (CPU-GPU Offloading). Training time is closely tied to the frequency of tensor offloading. More offloading increases training time, while a higher hit rate reduces it by serving more tensors directly from GPU cache. Fig. 15 compares the hit rates of 10Cache+P+M and ZeRO-Infinity, showing that 10Cache's optimized prefetch-eviction significantly improves GPU cache efficiency. Specifically, 10Cache+P+M achieves 27.17×, 4.87×, and 86.6× higher hit rates for OPT-6.7B, Bloom-7B, and Falcon-7B, respectively, than the baseline.

4.2.4 Training Time Evaluation (CPU-GPU-NVMe Offloading). Fig. 16 compares training time for OPT-13B, Falcon-10B, and Falcon-11B. 10Cache+FP16 schedules FP16 parameters while storing all optimizer states in NVMe. During the optimizer step, it sequentially loads optimizer states from NVMe to CPU memory for parameter updates. In contrast, 10Cache+FP16+Opt improves this by storing immediately required optimizer states in CPU memory (§ 3.4.2) and asynchronously fetching others during updates. This reduces training time by about 20% for OPT-13B, 23% for Falcon-10B, and 25% for Falcon-11B. ZeRO-Infinity suffers from performance inefficiencies due to its scheduling policy, which places all optimizer states in NVMe memory. Excess data offload leads to longer training times, highlighting the advantage of 10Cache's optimized scheduling.

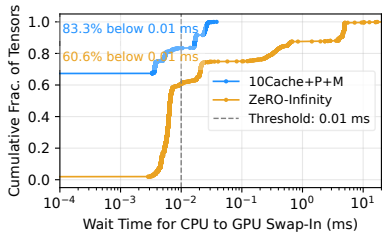
Table 5 compares per-iteration training times of 10Cache and baseline methods on the COLA dataset using the BERT [17] model with a batch size of 128 on an NVIDIA A40 GPU. 10Cache achieves up to a 19× speedup over prior GPU memory swapping approaches through efficient tensor caching and optimized prefetch-eviction. In the CPU–GPU–NVMe offloading setup (Table 5), 10Cache outperforms ZeRO-Infinity as its multi-tiered memory bandwidth-aware tensor allocator prioritizes GPU memory, selectively spills to CPU, and minimizes NVMe use. In contrast, ZeRO-Infinity places many tensors in NVMe, increasing migration latency, while 10Cache's optimized placement yields significant performance gains.

4.2.5 Wait Time Analysis for FP16 Parameters (CPU-GPU-NVMe). 10Cache+FP16+Opt processes 2.71× more tensors with wait times below 0.03 ms (§ 2.3) for Falcon-10B compared to ZeRO-Infinity (Fig. 17), with improvements of 1.91× for Falcon-11B and 1.18× for OPT-13B. Through strategic tensor placement, optimized prefetch-eviction, and efficient GPU cache use, 10Cache significantly reduces data offloading overhead, leading to efficient training execution.

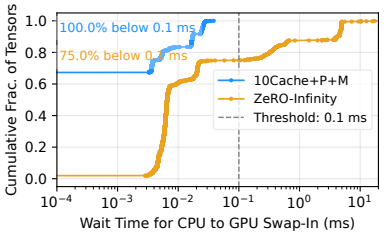


(a) OPT-6.7B (b) Bloom-7B (c) Falcon-7B

Figure 13: Model parameters wait time analysis for CPU-to-GPU transfer across models (OPT-6.7B, Bloom-7B, Falcon-7B).



(a) Threshold 0.01ms



(b) Threshold 0.1ms

Figure 14: OPT-6.7B parameters CPU-to-GPU transfer wait time at varying thresholds.

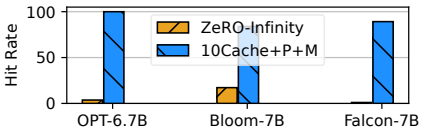


Figure 15: GPU cache hit rate comparison.

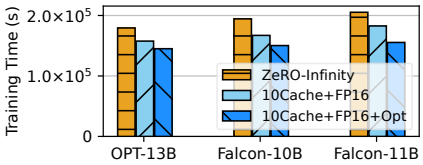


Figure 16: Training performance of 10Cache vs. ZeRO-Infinity under CPU-GPU-NVMe offloading.

4.2.6 FP16 Parameter Count Analysis in NVMe . The tensor allocator (§ 3.3.2) in 10Cache optimally places tensors across heterogeneous storage to minimize data offloading overhead. We evaluate its effectiveness by measuring the number of FP16 parameters stored in NVMe for 10Cache+FP16+Opt and the baseline in three models. 10Cache+FP16+Opt reduces the count in NVMe by 2.1× for OPT-13B, 6.7× for Falcon-10B, and 3.8× for Falcon-11B (Fig. 18), demonstrating that 10Cache's optimized tensor placement effectively reduces data offloading and improves training efficiency.

Table 5: Per-iteration training time comparison.

Approach	Per-Iter Time (ms)
FlashNeuron	5905.97
DeepUM	7566.76
G10	6667.69
10Cache (CPU-GPU Offloading)	381.70
ZeRO-Infinity (CPU-GPU Offloading)	462.79
10Cache (CPU-GPU-NVMe Offloading)	401.46
ZeRO-Infinity (CPU-GPU-NVMe Offloading)	7611.66

4.2.7 Cache Hit Rate and Miss Rate. In CPU-GPU-NVMe offloading, the FP16 parameter hit rate reflects GPU cache efficiency. Fig. 19a shows that 10Cache+FP16+Opt achieves up to 30.1× higher hit rate than ZeRO-Infinity through optimized prefetch-eviction and efficient GPU caching, while the baseline suffers from inefficient tensor scheduling.

The miss rate measures the effectiveness of 10Cache's optimizer state tensor scheduler. As shown in Fig. 19b, 10Cache+FP16+Opt keeps the miss rate below 1% in all models. This efficiency comes from caching selected optimizer states in CPU memory and asynchronously loading others from NVMe when needed (§ 3.4.2). In contrast, ZeRO-Infinity exhibits a much higher miss rate (around 100%), as it stores all optimizer states in NVMe and retrieves them synchronously, resulting in significant performance overhead.

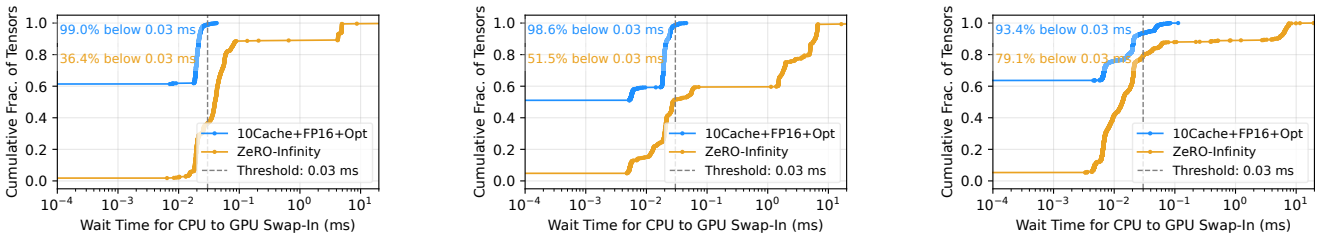
4.2.8 CPU-GPU Memory Utilization. We evaluate CPU and GPU memory utilization with 10Cache to understand how caching strategies, including memory allocation and tensor placement, improve CPU and GPU memory efficiency compared to the baseline.

CPU-GPU Offloading. Both ZeRO-Infinity and 10Cache store optimizer states in CPU, resulting in a small difference in CPU memory utilization. However, 10Cache improves CPU usage through pre-allocated cache memory for efficient offloading. As shown in Table 6, 10Cache achieves higher GPU memory utilization by caching more tensors than ZeRO-Infinity, leading to improved computational efficiency through better GPU memory management.

Table 6: Memory utilization (CPU-GPU offloading).

Model	10Cache (CPU Uti.)	ZeRO-Infinity (CPU Uti.)	10Cache (GPU Uti.)	ZeRO-Infinity (GPU Uti.)
OPT-6.7B	76%	70%	90%	76%
Bloom-7B	82%	72%	94%	78%
Falcon-7B	80%	72%	92%	78%

CPU-GPU-NVMe Offloading. Table 7 reveals a stark contrast in CPU memory utilization between the two approaches.



(a) Falcon-10B (b) Falcon-11B (c) OPT-13B

Figure 17: Model parameters wait time analysis for CPU-to-GPU transfer across models (Falcon-10B, Falcon-11B, OPT-13B).

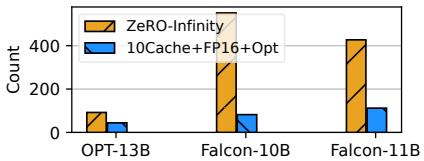
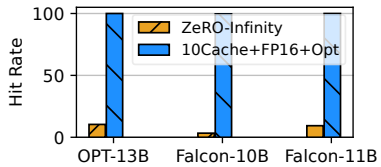
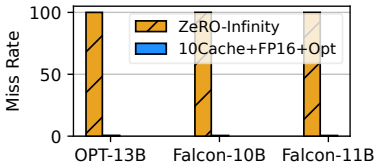


Figure 18: FP16 parameter count in NVMe across models.



(a) Hit rate (FP16 parameters).



(b) Miss rate (FP32 optimizer states).

Figure 19: Cache hit and miss rate comparison.

10Cache effectively utilizes CPU memory to store more tensors, while ZeRO-Infinity relies heavily on slower NVMe storage, resulting in up to 2.15× lower CPU memory utilization. 10Cache also achieves higher GPU memory utilization (up to 1.33×), further highlighting its ability to cache more tensors directly in GPU. In contrast, ZeRO-Infinity's inefficient scheduling limits memory optimization across CPU, GPU, and NVMe.

Table 7: Memory utilization (CPU-GPU-NVMe offloading).

Model	10Cache (CPU Uti.)	ZeRO-Infinity (CPU Uti.)	10Cache (GPU Uti.)	ZeRO-Infinity (GPU Uti.)
OPT-13B	82%	38%	90%	68%
Falcon-10B	76%	33%	94%	74%
Falcon-11B	76%	36%	88%	66%

4.2.9 Overhead Analysis of Profiling and Cache Memory Allocation. Table 8 reports the profiling and cache allocation times for different model sizes. For OPT-6.7B, 10Cache incurs only about 1% profiling time and 3.83% cache allocation time relative to a single training epoch. These results indicate that 10Cache adds minimal overhead, which becomes increasingly insignificant for larger models or longer training runs.

Table 8: Profiling and cache memory allocation time.

	Model Name	Profile Time (s)	Pre-allocate Cache Time (s)
CPU-GPU	OPT-6.7B	19.23 (1.09%)	67.26 (3.83%)
	Bloom-7B	19.96 (0.58%)	71.17 (2.07%)
	Falcon-7B	19.62 (0.19%)	70.84 (0.67%)
CPU-GPU-NVMe	Falcon-10B	108.54 (0.07%)	88.94 (0.06%)
	Falcon-11B	119.12 (0.08%)	87.09 (0.06%)
	OPT-13B	36.90 (0.03%)	69.83 (0.05%)

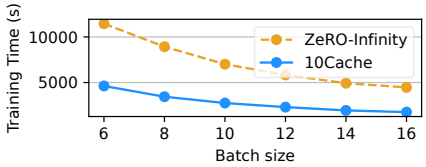


Figure 20: Training time for various batch sizes (OPT-6.7B).

4.2.10 Batch Size Impact on Training Time. Fig. 20 shows the OPT-6.7B model training time under CPU-GPU setup for varying batch sizes. As the batch size increases, 10Cache consistently reduces training time and outperforms ZeRO-Infinity.

In summary, our results show that 10Cache's optimal tensor placement maximizes memory usage, achieving up to 2.15× CPU and 1.33× GPU memory utilization, significantly outperforming the baseline (§ 4.2.8). Its smart caching with optimized prefetching and eviction boosts GPU cache hit rate by up to 86.6× (§ 4.2.3). Moreover, 10Cache adds minimal profiling overhead (§ 4.2.9), ensuring efficient integration into the training workflow.

5 RELATED WORK

GPUs enable efficient DNN training through high computational power, but the rapid increase in model size makes GPU memory a major bottleneck [39]. Prior works [22, 24, 62] mitigate this by extending GPU memory with CPU-based swapping, mainly for CNNs, but performance degrades when the CPU handles memory-intensive tasks. FlashNeuron [30] offloads intermediate tensors to SSDs to reduce CPU load, yet NVMe access latency remains a challenge. Sentinel [50] dynamically profiles tensors using TensorFlow runtime and OS page faults. It maps page-level profiling to tensors by assigning each tensor a dedicated memory page and adding layer-end annotations, which greatly increase memory footprint and profiling overhead, issues that might grow with LLMs. In contrast, 10Cache performs a single lightweight profiling iteration using PyTorch hooks to capture tensor execution order directly, making

it more efficient and robust. While Sentinel migrates long-lived tensors based on memory access frequency, 10Cache prefetches/evicts tensors by execution order, keeping needed tensors in fast memory and preventing GPU stalls. Another line of works use CUDA Unified Memory with page prefetching [21, 25]. DeepUM [25] profiles memory access patterns via GPU page faults to improve memory management but remains limited by CPU and GPU capacity, hindering scalability for large models. G10 [21] unifies CPU, GPU, and flash memory and supports page-level tensor migration. However, these methods overlook the distinct memory behavior of LLMs (§ 2.1). 10Cache addresses this by managing memory with pre-allocated cache buffers and fine-grained tensor-level migration across CPU, GPU, and NVMe based on execution order, enabling efficient training of billion-parameter LLMs.

Recent studies [29, 48, 49, 57, 59] propose memory management techniques to scale LLM training. ZeRO-Offload [29] reduces GPU memory use by offloading gradients and optimizer states to CPU but lacks parameter offloading. ZeRO-Infinity [53] adds NVMe offloading but its inefficient tensor placement across CPU, GPU, and NVMe leads to suboptimal memory utilization. In contrast, 10Cache offers finer memory-aware placement across all tiers, improving memory utilization (table 6, table 7) and GPU cache hit rates (§ 4.2.3, § 4.2.7). L2L [48] keeps only the active layer in GPU, incurring a high CPU-GPU communication cost, while StrongHold [59] manages a sliding window of active layers in GPU that requires window size tuning. Poor tuning underutilizes GPU memory and increases layer transfer overhead. In contrast, 10Cache dynamically places tensors based on memory availability, eliminating tuning and reducing data movement through efficient caching and scheduling, thereby maximizing memory efficiency and minimizing migration overhead. SHADE [35] and FedCaSe [34] apply importance-aware sampling and caching to scale deep and federated learning but focus only on in-memory caching for computer-vision workloads. In contrast, 10Cache optimizes LLM fine-tuning via fine-grained tensor migration across a multi-tier cache hierarchy.

Arif et al. [13] offload optimizer states to CXL memory but not parameter tensors (up to 58% of memory, Fig. 2), limiting performance in resource-constrained settings and adding communication overhead [27]. MemAscend [42] addresses SSD-to-GPU transfer bottlenecks due to CPU buffer pool and internal fragmentation of fixed-size buffers using an adaptive pool. In contrast, 10Cache addresses these issues with a smaller buffer pool and efficient memory management. Both works lack tensor lifetime analysis and CPU/GPU cache reuse, which are critical in constrained environments. While Mixture-of-Experts (MoE) and sparsity reduce memory usage, they add complexity (e.g., load balancing and communication overhead) [46, 63]. 10Cache focuses on dense transformers and complements sparsity by managing memory for both active and selectively used tensors. ZenFlow [40] is a recent training framework that offloads GPU memory by prioritizing parameters and splitting updates between the GPU and CPU to reduce GPU stalls and I/O overhead. SuperOffload [41] introduces a Superchip-centric [7] system that efficiently leverages the Hopper GPU, Grace CPU, and NVLink-C2C interconnect through adaptive weight offloading, bucket repartitioning, and an optimized Adam optimizer. Both ZenFlow and SuperOffload are orthogonal to 10Cache and can be integrated with it to further enhance training throughput.

6 DISCUSSION

10Cache significantly reduces training time for billion-scale LLMs through efficient memory management, though parts of its design still require further exploration and refinement.

Dynamic Execution Graph. 10Cache targets LLM workloads with static execution graphs, where operation order and tensor access patterns are predictable and repeated across training iterations. This predictability enables 10Cache to precompute prefetching and eviction schedules. Although dynamic execution graphs [15, 45, 51, 66] are gaining attention, 10Cache does not yet support them due to their unpredictable nature. Future work will focus on extending 10Cache to learn and adapt to dynamic execution patterns at runtime for optimized prefetching and scheduling.

Distributed Environment. 10Cache explicitly targets the single-GPU setup, which is highly relevant for resource-constrained cloud users who cannot afford large GPU clusters. Extending 10Cache to distributed training (e.g., pipeline or tensor parallelism) is a key future direction. In such settings, network communication and resource contention can make execution less predictable. To adapt, 10Cache's prefetch table can be extended to record tensor execution order and GPU ownership, enabling the scheduler to prefetch and evict based on memory hierarchy and inter-GPU communication.

Inference. The techniques proposed in 10Cache have not yet been evaluated alongside established methods such as prefill/decode optimization, KV-cache management, or workload-aware request scheduling [28, 33, 38, 55, 58, 67]. In future work, we will integrate our mechanisms, e.g., efficient tensor offloading, into a unified LLM serving architecture optimized for constrained-memory settings.

7 CONCLUSION

We present 10Cache, a novel framework that accelerates billion-scale LLM training for the GPU memory swapping mechanism. It features a lightweight profiler that analyzes tensor execution order to optimize placement across heterogeneous memory tiers, maximizing memory utilization, and leveraging the full memory hierarchy to boost training throughput. 10Cache improves performance through efficient cache allocation, reuse of memory buffers, and intelligent prefetching and eviction that overlap data transfers with GPU computation. Experimental results show that 10Cache reduces training time by about 2×, increases GPU cache hit rate by up to 86.6×, and achieves CPU and GPU memory utilization by 2.15× and 1.33×, respectively. These results show that 10Cache effectively alleviates memory bottlenecks, accelerates LLM training, and provides a practical solution for cloud environments.

8 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. Some results were obtained using the Chameleon testbed [32], supported by NSF. This work is supported in part by NSF grants CSR-2106634 and CSR-2312785.

REFERENCES

[1] 2012. Pinned Host Memory. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/.
[2] 2017. NVIDIA Tesla V100 GPU architecture. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[3] 2017. NVIDIA. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.
[4] 2025. DeepSpeed. https://github.com/microsoft/DeepSpeed/tree/master.
[5] 2025. Fx Graph. https://pytorch.org/docs/stable/fx.html.
[6] 2025. GPUDirect Storage. https://developer.nvidia.com/blog/gpudirect-storage/.
[7] 2025. NVIDIA GH200 Grace Hopper Superchip. https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/.
[8] 2025. NVIDIA H100 GPU. https://www.nvidia.com/en-eu/data-center/h100/.
[9] 2025. PyTorch. https://pytorch.org/.
[10] 2025. PyTorch Hook. https://pytorch.org/docs/stable/generated/torch.nn.Module.html.
[11] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, and et al. 2023. Gpt-4 technical report. arXiv:2303.08774 [cs.CV]
[12] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, and Étienne Goffinet et al. 2023. The falcon series of open language models. arXiv:2311.16867 [cs.CV]
[13] Moiz Arif, Avinash Maurya, Sudharshan Vazhkudai, and Bogdan Nicolae. 2025. Evaluating Expansion Memory for Optimizer State Offloading for Large Transformer Models. In *HPAI4S'25: HPC for AI Foundation Models & LLMs for Science (co-located with IPDPS'25)*.
[14] David Chappell. 2010. Introducing the windows azure platform. In *David Chappell & Associates White Paper*.
[15] Simin Chen, Shiyi Wei, Cong Liu, and Wei Yang. 2023. Dycl: Dynamic neural network compilation via program rewriting and graph optimization. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 614–626.
[16] Esha Choukse, Michael B. Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David Nellans, and Stephen W. Keckler. 2020. Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 926–939.
[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*. 2.
[18] Xinwei Fu, Zhen Zhang, Haozheng Fan, Guangtai Huang, Mohammad El-Shabani, and Randy Huang et al. 2024. Distributed training of large language models on aws trainium. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. 961–976.
[19] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, and Minyi Guo et al. 2024. Gmlake: Efficient and transparent gpu memory defragmentation for large-scale dnn training with virtual memory stitching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 450–466.
[20] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning*. 1737–1746.
[21] Zhang Haoyang, Zhou Yirui, Xue Yuqi, Liu Yiqi, and Huang Jian. 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.
[22] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 875–890.
[23] Jeremy Howard and Sebastian Ruder. 2018. Universal Language Model Finetuning for Text Classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*. 328–339.
[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. https://doi.org/10.1145/3373376.3378530
[25] Jung Jaehoon, Kim Jinpyo, and Lee Jaejin. 2023. DeepUM: Tensor Migration and Prefetching in Unified Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 207–221.
[26] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. 2018. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 776–789.
[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. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 345–360.
[28] Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Taiyi Wang, Bin Cui, and Ana Klimovic et al. 2025. Thunderserve: High-performance and cost-efficient llm serving in cloud environments. arXiv:2502.09334 [cs.CV]
[29] Ren Jie, Rajbhandari Samyam, Aminabadi Reza Yazdani, Ruwase Olatunji, Yang Shuangyan, Zhang Minjia, Li Dong, and He Yuxiong. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *In 2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.
[30] Bae Jonghyun, Lee Jongsung, Jin Yunho, Son Sam, Kim Shine, Jang Hakbeom, Ham Tae Jun, and Lee Jae W. 2021. FlashNeuron:SSD-Enabled Large-Batch Training of Very Deep Neural Networks. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 387–401.
[31] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–12.
[32] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, and Dan Stanzione et al. 2020. Lessons learned from the chameleon testbed.. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 219–233.
[33] Redwan Ibne Seraj Khan, Kunal Jain, Haiying Shen, Ankur Mallick, Anjaly Parayil, Anoop Kulkarni, Steve Kofsky, Pankhuri Choudhary, Renee St Amant, Rujia Wang, et al. 2024. Ensuring Fair LLM Serving Amid Diverse Applications. *arXiv preprint arXiv:2411.15997* (2024).
[34] Redwan Ibne Seraj Khan, Arnab K Paul, Yue Cheng, Xun Steve Jian, and Ali R Butt. 2024. FedCaSe: Enhancing Federated Learning with Heterogeneity-aware Caching and Scheduling. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. 52–68.
[35] Redwan Ibne Seraj Khan, Ahmad Hossein Yazdani, Yuqi Fu, Arnab K. Paul, Bo Ji, Xun Jian, Yue Cheng, and Ali R. Butt. 2023. SHADE: Enable Fundamental Cacheability for Distributed Deep Learning Training. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 135–152. https://www.usenix.org/conference/fast23/presentation/khan
[36] Junkyum Kim, Myeonggu Kang, Yunki Han, Yang-Gon Kim, and Lee-Sup Kim. 2023. Optimstore: In-storage optimization of large scale dnns with on-die processing. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 611–623.
[37] Diederik P. Kingma and Ba Jimmy. 2015. Adam: A method for stochastic optimization. In *ICLR*.
[38] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, and Cody Hao Yu et al. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.
[39] Youngeun Kwon and Minsoo Rhu. 2018. Beyond the memory wall: A case for memory-centric hpc system for deep learning. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 148–161.
[40] Tingfeng Lan, Yusen Wu, Bin Ma, Zhaoyuan Su, Rui Yang, and Tekin Bicer et al. 2025. ZenFlow: Enabling Stall-Free Offloading Training via Asynchronous Updates. arXiv:2505.12242 [cs.CV]
[41] 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.CV]
[42] Yong-Cheng Liaw and Shuo-Han Chen. 2025. MemAscend: System Memory Optimization for SSD-Offloaded LLM Fine-Tuning. arXiv:2505.23254 [cs.CV]
[43] Sajee Mathew and J. Varia. 2014. Overview of amazon web services. In *Amazon Whitepapers 105*. 22.
[44] Paulius Micikevicius, Sharan Narang, Gregory Diamos Jonah Alben, Erich Elsen, David Garcia, and Boris Ginsburg et al. 2017. Mixed precision training. arXiv:1710.03740 [cs.CV]
[45] Wei Niu, Gagan Agrawal, and Bin Ren. 2024. SoD2: Statically Optimizing Dynamic Deep Neural Network Execution. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 386–400.
[46] Xinglin Pan, Wenxiang Lin, Lin Zhang, Shaohuai Shi, Zhenheng Tang, and Rui Wang et. al. 2025. FSMoE: A Flexible and Scalable Training System for Sparse Mixture-of-Experts Models. arXiv:2501.10714 [cs.CV]
[47] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, and Nithish Mahalingam et al. 2024. Characterizing power management opportunities for llms in the cloud. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 207–222.
[48] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. 2020. Training large neural networks with constant memory using a new execution algorithm. arXiv:2002.05645 [cs.CV]
[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*. 1–16. https://doi.org/10.1109/SC41405.2020.00024
[50] 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. https://doi.org/10.1109/

HPCA51647.2021.00057

[51] Jie Ren, Dong Xu, and Shuangyan Yang et al. 2024. Enabling Large Dynamic Neural Network Training with Learning-based Memory Management. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 788–802.

[52] Agniswar Roy, Abhik Banerjee, and Navneet Bhardwaj. 2021. A study on Google Cloud Platform (GCP) and its security. In *Machine Learning Techniques and Analytics for Cloud Security*. 313–338.

[53] Rajbhandari Samyam, Ruwase Olatunji, Rasley Jeff, Smith Shaden, and He Yuxiong. 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*. 1–14.

[54] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, et al. 2023. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. (Nov. 2023). https://inria.hal.science/hal-03850124 working paper or preprint.

[55] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, and Beidi Chen et al. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*. 31094–31116.

[56] Aditya S. Shethiya. 2025. Deploying AI Models in. NET Web Applications Using Azure Kubernetes Service (AKS). In *Spectrum of Research 5*.

[57] 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:1909.08053 [cs.CV]

[58] Qidong Su, Wei Zhao, Xin Li, Muralidhar Andoorveedu, Chenhao Jiang, and Zhanda Zhu et al. 2025. Seesaw: High-throughput LLM Inference via Model Re-sharding. In *Proceedings of Machine Learning and Systems*.

[59] Xiaoyang Sun, Wei Wang, S. Qiu, R. Yang, S. Huang, J. Xu, and Z. Wang. 2022. STRONGHOLD: Fast and Affordable Billion-Scale Deep Learning Model Training. In *SC'22: The International Conference for High Performance Computing, Networking, Storage and Analysis. Association for Computing Machinery*.

[60] Robert Tinn, Hao Cheng, Yu Gu, Naoto Usuyama, Xiaodong Liu, Tristan Naumann, Jianfeng Gao, and Hoifung Poon. 2023. Fine-tuning large neural language models for biomedical natural language processing. Patterns. In *Patterns 4, no. 4.*

[61] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, and Baptiste Rozière et al. 2023. Llama: Open and efficient foundation language models. arXiv:2302.13971 [cs.CV]

[62] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, and Ang Li et al. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 41–53.

[63] Chenpeng Wu, Qiqi Gu, Heng Shi, Jianguo Yao, and Haibing Guan. 2025. Samoyeds: Accelerating MoE Models with Structured Sparsity Leveraging Sparse Tensor Cores. In *Proceedings of the Twentieth European Conference on Computer Systems*. 293–310.

[64] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, and Christopher Dewan et al. 2022. Opt: Open pre-trained transformer language models. arXiv:2205.01068 [cs.CV]

[65] Pinxue Zhao, Hailin Zhang, Fangcheng Fu, Xiaonan Nie, Qibin Liu, and Fang Yang an Yuanbo Peng et al. 2025. MEMO: Fine-grained Tensor Management For Ultra-long Context LLM Training. In *Proceedings of the ACM on Management of Data 3, no. 1.* 1–28.

[66] Bojian Zheng, Cody Hao Yu, Jie Wang, Yaoyao Ding, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 2023. Grape: Practical and Efficient Graphed Execution for Dynamic Deep Neural Networks on GPUs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1364–1380.

[67] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, and Xuanzhe Liu et al. 2024. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 193–210.