TRANSIT: Transparent Scale-in for Multi-Node LLM Training

Hyungyo Kim[1†], Nicholas Satchanov[1], Hrishi Shah[1], Gaohan Ye[1], Jiaqi Lou[1],
Robert Walkup[2], Shweta Salaria[2], I-Hsin Chung[2], Hubertus Franke[2],
Seetharami Seelam[2], Apoorve Mohan[2*], Nam Sung Kim[1*]

[1]University of Illinois at Urbana-Champaign [2]IBM Research

arXiv:2610.07593v2 [cs.DC] 19 Nov 2026

Abstract

The growing memory requirements of large language models (LLMs) increasingly force distributed training jobs to span multiple GPU nodes to accommodate their training state. This reduces per-GPU computational efficiency, increases network traffic, and exacerbates job queueing delays. To address these inefficiencies, we introduce TRANSIT, a system that transparently enables efficient distributed LLM training on fewer GPUs through memory oversubscription. TRANSIT optimizes CPU–GPU data movement during oversubscription by coordinating driver-managed page residency with execution-guided zero-copy access, exploiting recurring kernel execution patterns across training iterations.

We evaluate TRANSIT on dense and mixture-of-experts (MoE) models across different parallelization strategies using up to 64 NVIDIA H100-80GB GPUs interconnected over RDMA over Converged Ethernet (RoCE). Compared with the minimum GPU count required to accommodate the training state in GPU memory, TRANSIT enables training on up to 50% fewer GPUs while retaining over 90% of the baseline per-GPU computational throughput. In some configurations, training on 17–33% fewer GPUs even improves throughput due to reduced distributed communication overhead. When training on fewer GPUs, TRANSIT achieves up to 243% and 87% higher throughput than conventional driver-runtime-managed memory oversubscription and framework-managed CPU offloading, respectively, while reducing CPU memory consumption by up to 79% compared with the latter. Training on fewer GPUs with TRANSIT also reduces per-node network traffic by up to 33% and improves throughput by up to 44% under constrained network bandwidth. Production-trace-driven simulations further show 15–49% lower inter-rack network traffic and up to 2–95% shorter job queue times for different cluster capacities.

1 Introduction

Forced Memory-Driven Scale-out. The cost of training large language models (LLMs) has grown to the point where it shapes fundamental decisions about computing infrastructure and resource provisioning [9]. For many training configurations, GPU allocations depend not only on computational requirements but also on the memory capacity needed to store parameters, gradients, optimizer states, and activations [16]. These demands can exceed the aggregate capacity of GPU high-bandwidth memory (HBM) available on a modest GPU allocation. For example, training Llama4-Scout [28] in our evaluated configuration (§5.1) requires 64 NVIDIA H100-80GB GPUs to accommodate its training state.

Scale-out incurs well-known inefficiencies. For example, it limits (I1) *training feasibility*: academic labs, startups, and enterprises operating under budget or infrastructure constraints may be unable to provision the required GPUs, while even well-funded organizations face availability, procurement, and supply-chain constraints [12]. It also introduces (I2) *computational inefficiency*: increasing model parallelism to accommodate training state can partition computation into smaller operations and add communication and synchronization overhead, yielding disproportionately small throughput gains from additional GPUs [14]. At the datacenter level, larger distributed configurations increase (I3) *network sensitivity* to bandwidth limitations, contention, and delays [10, 17, 35, 39, 41, 51]; expand the (I4) *failure domain*, increasing exposure to GPU, node, and interconnect failures [24]; and exacerbate (I5) *cluster resource inefficiency*, as fixed-size GPU allocations can experience substantial queueing delays under resource fragmentation and capacity constraints [19, 21, 32, 47].

These inefficiencies motivate "scale-in", *i.e.,* enabling distributed training jobs to execute on fewer GPUs than their memory requirements would otherwise dictate. By reducing memory-driven GPU allocations, scale-in can improve GPU resource efficiency, reduce communication demand, increase cluster-scheduling flexibility, and improve reliability from smaller distributed configurations.

Challenge: Oversubscription Without Losing Throughput. Scale-in requires accommodating training state that exceeds the reduced aggregate GPU HBM capacity. Modern GPU servers offer an opportunity through substantial CPU DRAM capacity, which can serve as an additional memory tier. However, CPU DRAM has substantially lower bandwidth and higher access latency than HBM, and inefficient data movement over PCIe can impact training performance. The central challenge is therefore to oversubscribe GPU memory using CPU DRAM while preserving per-GPU training throughput; otherwise, memory-management overheads can negate the benefits of scale-in.

A major source of overhead is the CPU–GPU data movement during oversubscription. Driver-runtime-managed oversubscription (such as CUDA Unified Virtual Memory (UVM) [7] and HIP Managed Memory [1]) transparently migrates pages on-demand without application-level knowledge of access and reuse patterns, incurring page-fault overheads and repeated migration–eviction cycles that amplify PCIe traffic. These costs are high enough that PyTorch's documentation discourages using UVM for deep learning workloads [4]. Framework-managed offloading (such as implemented by DeepSpeed [36–38], TorchTitan [26], and Megatron-LM [31, 40]) leverages application-level knowledge to orchestrate tensor-level

[†]This work was initiated during Hyungyo Kim's internship at IBM Research and was subsequently continued through the IBM-Illinois Discovery Accelerator Institute (IIDAI).

[*]Corresponding authors: Apoorve Mohan (Apoorve.Mohan@ibm.com) and Nam Sung Kim (nam.sung.kim@gmail.com).

transfers, but shifts memory-management complexity to developers, requires framework-specific integration, and can incur substantial PCIe traffic-related overheads. Importantly, despite their different memory-management strategies, both approaches can incur substantial CPU–GPU data movement overhead, either through repeated page migrations under memory pressure or through coarse-grained tensor transfers. In our experiments, we observed that these memory oversubscription approaches reduce per-GPU training throughput by up to about 38% when scaling Llama3-70B training from 24 to 16 NVIDIA H100-80GB GPUs (see §5.2).

The underlying challenge, thus, is balancing the benefits of HBM residency against the migration overhead during a training iteration. Migrating CPU-resident data into HBM incurs a one-time transfer cost that subsequent accesses amortize through HBM's higher bandwidth and lower latency; however, it consumes limited capacity and can evict useful pages. Conversely, direct CPU-memory access avoids migration but pays higher latency and PCIe cost on every access. Existing approaches [23, 27] primarily optimize which data to move and when, rather than avoiding migration through execution-guided direct access. This raises the following **research questions:** *- Can recurring kernel execution provide sufficient guidance to decide when to enable zero-copy access to CPU-resident data rather than relying on page migration? - Can such an approach mitigate memory oversubscription overhead without explicit data placement, prefetching, or fine-grained memory-access tracking? - Can efficient memory oversubscription enable distributed LLM training on fewer GPUs while preserving high per-GPU computational efficiency?*

Our Approach. We present **TRANSIT** (TRANsparent Scale-In for multi-node Training), a transparent GPU memory oversubscription system that coordinates driver-managed page residency with execution-guided zero-copy access to CPU-resident data, enabling efficient scale-in for distributed LLM training. By leveraging CPU DRAM as an extension of GPU HBM, TRANSIT reduces memory-driven GPU requirements while preserving per-GPU training throughput. Unlike approaches that use tensor lifetimes [49, 50] or page-access correlations [23] to predict and schedule data movement, TRANSIT exploits recurring kernel-launch patterns to guide zero-copy activation without tracking individual memory accesses. The GPU driver manages the page residency, while TRANSIT uses coarse-grained execution progress to decide when to enable zero-copy of CPU-resident pages. This allows efficient memory oversubscription without tensor-level knowledge, page-access profiling, explicit data-placement decisions, or framework integration.

Observations. TRANSIT's design is motivated by three observations about the limitations of migration-centric oversubscription and the tradeoffs of direct CPU-memory access (see §3 for details). (O1) First, optimizing fine-grained migrations improves PCIe transfer efficiency but does not eliminate the cost of moving CPU-resident pages into HBM. (O2) Second, under memory pressure, repeated migration and eviction can generate substantial data movement, which zero-copy access can avoid by accessing CPU-resident pages directly. (O3) Third, naive zero-copy can significantly degrade performance because frequently accessed data benefits from HBM residency. However, we find that a broad range of zero-copy activation points preserves performance during training iterations, suggesting that recurring kernel-launch progress can guide when to enable zero-copy without tracking individual memory accesses or reconstructing tensor lifetimes.

Design. These observations show that the timing of zero-copy activation determines whether direct CPU-memory access complements or undermines the GPU driver's residency decisions. Before activation, on-demand migration lets frequently accessed pages (*i.e.,* hot pages) establish HBM residency; after activation, the GPU can access CPU-resident pages (*i.e.,* cold pages) directly through its L2 cache without migrating them into HBM. TRANSIT realizes this with three user-space components (Figure 7): an allocation interceptor that redirects GPU allocations to managed memory, an activity recorder that counts kernel launches per iteration during warm-up, and a zero-copy controller that enables direct access for the relevant managed-memory regions once kernel-launch progress reaches the derived activation threshold (see §4 for details). The advice is applied once: pages resident in HBM at activation stay local, and pages in CPU memory are thereafter accessed in place rather than migrated. Activation thus preserves the existing access paths for HBM-resident pages while enabling direct access to CPU-resident pages, and the kernel-launch threshold determines when these mappings are established. Each GPU acts independently, adding no inter-GPU communication or synchronization, and TRANSIT requires no modifications to training applications, frameworks, communication libraries, schedulers, or device drivers.

This paper makes the following **contributions**:

(C1) We characterize the limitations of GPU memory oversubscription for distributed LLM training, demonstrating that improving fine-grained migration efficiency alone is insufficient under memory pressure. We quantify the costs of migration-induced data movement and the performance sensitivity of zero-copy activation timing, establishing the need to coordinate HBM residency with direct CPU-memory access.

(C2) We design and implement TRANSIT, a lightweight GPU memory oversubscription system that exploits recurring kernel-launch patterns to guide zero-copy activation while retaining driver-managed page residency, without tensor-lifetime analysis, page-access profiling, or explicit data-placement decisions. Implemented as a user-space shared library, TRANSIT enables transparent scale-in across training configurations without framework-managed offloading or modifications to the training stack.

(C3) We evaluate TRANSIT on dense and mixture-of-experts LLMs across multiple parallelism configurations using up to 64 NVIDIA H100 GPUs interconnected over RoCE. Our evaluation shows up to 87% higher per-GPU throughput than state-of-the-art framework-managed approaches, and up to 243% higher per-GPU throughput than conventional driver-runtime-managed oversubscription (CUDA UVM). TRANSIT enables training with up to 50% fewer GPUs while maintaining over 90% of baseline per-GPU throughput in the evaluated configurations. Scale-in reduces per-node network traffic by up to 33% and improves per-GPU training throughput by up to 44% in network-constrained environments. Production-trace-driven simulations further demonstrate reductions of 15–49% in inter-rack network traffic and up to 2–95% in job queueing time for different cluster capacities.

2 The Case for Scale-In

Distributed LLM training must balance computational efficiency, memory capacity, and communication overhead across GPUs and nodes. However, the GPU allocation needed to accommodate training state may differ from what efficient computation requires, creating opportunities for scale-in. This section reviews the factors behind memory-driven scale-out, its inefficiencies, and the conditions that motivate scale-in.

2.1 Memory-Driven Scale-Out and Its Costs

Memory requirements and parallelism. GPU memory must accommodate model parameters, gradients, optimizer states, and activations during LLM training [36]. Mixed-precision training [29], activation checkpointing [11], and memory-efficient attention such as FlashAttention [13] reduce this footprint, yet substantial memory requirements remain. Our Llama4-Scout [28] configuration (global batch size 64, sequence length 8192) requires about 4.3 TB of aggregate GPU memory even with these optimizations, and therefore 64 NVIDIA H100-80GB GPUs in our evaluated configuration without offloading. Training frameworks distribute this state by combining tensor parallelism (TP) [40], pipeline parallelism (PP) [22, 30], fully sharded data parallelism (FSDP) [26] or ZeRO [36], and, for mixture-of-experts (MoE) models, expert parallelism (EP) [25, 44]. Because partitioning training state also determines how computation and communication are distributed, increasing parallelism to meet memory constraints can reduce local computation while adding communication and synchronization overhead. Memory capacity can thus impose a minimum GPU allocation that differs from the one that uses computational resources most efficiently.

Computational inefficiency. Memory-driven scale-out can reduce computational efficiency when additional GPUs are allocated primarily to accommodate training state rather than increase useful computation. Increasing the degree of model parallelism can partition computation into smaller operations, reducing local work while adding communication and synchronization overhead [14, 52]. As training spans multiple nodes, collective communication becomes increasingly dependent on inter-node bandwidth and network conditions [31, 40]. Consequently, additional GPUs provisioned to satisfy memory requirements may contribute disproportionately little to aggregate training throughput. The extent of this inefficiency depends on the workload and parallelization strategy, as increasing GPU count does not necessarily reduce computational efficiency in every configuration.

Network sensitivity and stragglers. Distributed LLM training relies on collective communication, including all-gather, reduce-scatter, and all-to-all operations, to exchange model states, gradients, and intermediate results. These operations introduce synchronization dependencies, making training performance sensitive to network bandwidth, congestion, and slow participants [46]. In shared datacenters, network oversubscription and contention can further reduce effective bandwidth and introduce performance variability [15, 34, 41]. Prior studies of production AI training clusters report network contention affecting up to approximately 36% of training jobs [10], with congestion-related slowdowns exceeding 30% in some settings [35]. We observe a similar straggler effect in our experiments. When training Llama4-Scout on 64 NVIDIA H100 GPUs across 16 nodes, reducing a single node's NIC bandwidth from 400 Gbps to 100 Gbps decreases overall training throughput by 50%: because collectives synchronize participants, the slowest node bounds the job's progress. Larger configurations involve more nodes and links that can degrade, increasing exposure to stragglers.

Cluster scheduling and reliability. Memory-driven scale-out also affects resource allocation at the datacenter level. Distributed training jobs commonly request a fixed number of GPUs that must be allocated before execution begins [32, 47]. Under capacity constraints, larger allocations can experience prolonged queueing delays. Even when sufficient aggregate GPU capacity is available, resource fragmentation and topology-aware placement requirements can prevent a scheduler from satisfying a job's allocation request [21]. This is particularly consequential in development and experimentation environments, where queueing delays can limit iteration speed. Larger distributed configurations also involve more GPUs, nodes, and interconnect components, expanding the failure domain and increasing exposure to component failures and recovery overheads [24]. Thus, provisioning additional GPUs primarily for memory capacity can impose costs beyond the computational resources the job consumes, including network sensitivity, scheduling constraints, and reliability risks.

2.2 Scale-In Opportunity

Reducing memory-driven GPU allocations. The inefficiencies of memory-driven scale-out motivate an alternative approach: reducing the number of GPUs required for training by leveraging CPU DRAM as an additional memory tier. By moving part of the training state outside GPU HBM, scale-in increases the effective memory capacity per allocated GPU without adding GPUs primarily for memory. At the job level, scale-in can make training feasible on infrastructure that cannot accommodate the original GPU allocation. At the datacenter level, smaller allocations can improve scheduling flexibility, reduce resource fragmentation, and allow more jobs to execute concurrently under capacity constraints. Reducing the number of participating GPUs and nodes can also lower communication demand, although the effect depends on the parallelization strategy and workload. In particular, reducing GPU count does not necessarily reduce the communication volume of every collective operation.

Local CPU DRAM versus remote GPU HBM. When collectives span nodes, data movement depends on network bandwidth, latency, synchronization, and contention. In contrast, node-local CPU DRAM provides additional memory capacity without inter-node communication, albeit at lower bandwidth and higher latency than local GPU HBM. Figure 1 compares local CPU DRAM access over PCIe Gen5 with remote GPU HBM access over a 400 Gbps ConnectX-7 network using GPUDirect RDMA (GDR) [2, 5]: local access achieves 5–6× lower latency for small transfers and 2–3× lower latency for larger transfers. These measurements highlight an opportunity for scale-in: local CPU DRAM can provide a competitive memory-access path relative to remote GPU transfers, enabling jobs to run on fewer nodes, potentially reducing distributed communication, and thus alleviating network pressure in bandwidth-constrained, congested, or oversubscribed environments.

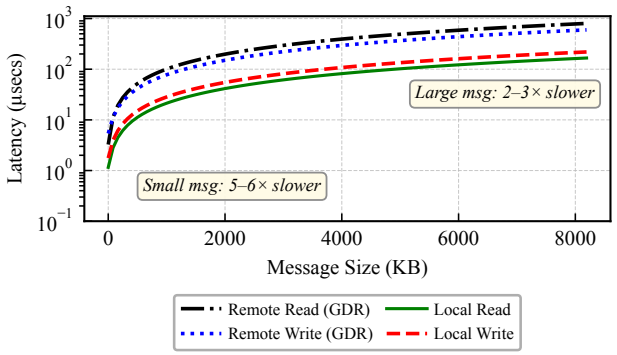

Figure 1: Read and write latencies for accessing local CPU DRAM over PCIe compared to remote GPU HBM accessed via GPUDirect RDMA (GDR) on NVIDIA H100 systems.

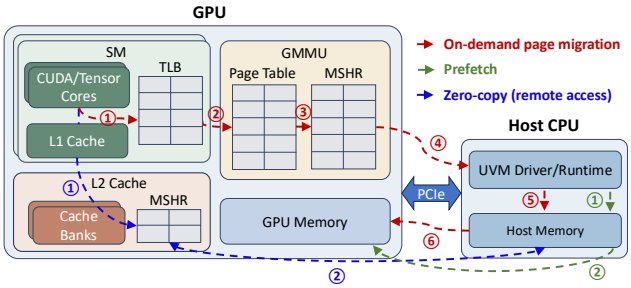

Figure 2: UVM data access paths: on-demand migration (red), prefetch (green), and zero-copy (blue).

Performance and resource tradeoffs. Scale-in consolidates training state and computation onto fewer GPUs, increasing per-device memory requirements. Using CPU DRAM as additional memory introduces access overhead relative to local HBM, while reducing GPU count can lower aggregate throughput and increase job completion time even when per-GPU efficiency is preserved. Nevertheless, smaller allocations can reduce queueing delays and network footprint while improving cluster scheduling flexibility. Importantly, fewer GPUs reduce total GPU-hours only if per-GPU throughput exceeds that of the original allocation. Thus, the objective is to reduce memory-driven GPU allocations while limiting the performance overhead of accessing training state outside HBM. We next examine GPU memory oversubscription mechanisms and LLM training memory-access patterns that determine this overhead.

2.3 Memory Access and Management

GPU memory oversubscription. GPU memory oversubscription lets a workload's memory footprint exceed HBM capacity by leveraging larger-capacity CPU DRAM, whose access over PCIe incurs higher latency and lower bandwidth. Its efficiency depends on how data moves between these memory tiers and how often the GPU accesses CPU-resident data. Framework-managed offloading, as in DeepSpeed [37, 38] and TorchTitan [26], places

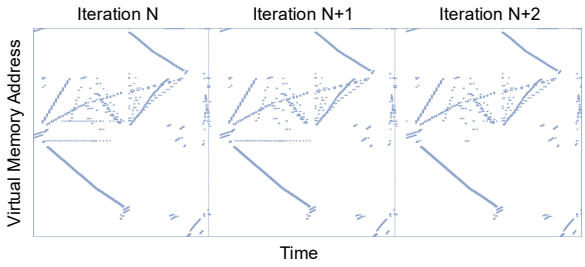

Figure 3: Data migration pattern during training of Llama3-8B on an NVIDIA H100 NVL (94 GB HBM), with a memory footprint at 145% of HBM capacity.

selected training state in CPU memory and exploits tensor lifetimes and training dependencies to schedule transfers and overlap them with computation. However, it requires framework-level coordination, typically manages transfers at tensor granularity, and its performance depends on transfer volume and computation overlap. In contrast, CUDA Unified Virtual Memory (UVM) [7] provides driver-runtime-managed page residency, enabling transparent, page-granular oversubscription without explicit framework orchestration. Under demand-driven migration, GPU accesses to nonresident managed pages can trigger page faults and migrations from CPU DRAM into HBM (Figure 2), incurring fault-handling and transfer overhead and potentially causing repeated migration–eviction cycles under HBM pressure. Alternatively, *zero-copy* allows the GPU to access CPU-resident data directly over PCIe without first migrating it into HBM. While migration incurs an upfront transfer cost that can be amortized through subsequent HBM accesses, zero-copy avoids migration but incurs remote-memory access overhead. Their relative efficiency therefore depends on transfer overhead, data reuse, and HBM residency.

Memory-access and kernel-execution patterns. LLM training exhibits structured spatial and temporal locality. Spatial locality arises from contiguous tensor accesses, while temporal locality reflects reuse of training state across execution phases. During the forward pass, parameters are accessed layer by layer to compute activations, which are reused during backward propagation. Parameters and gradients may be accessed repeatedly, particularly with gradient accumulation, whereas optimizer states and higher-precision parameter copies are accessed mainly during optimizer updates, resulting in longer reuse intervals. Figure 3 illustrates these characteristics through UVM page-migration patterns over three iterations of Llama3-8B training on a single H100 NVL GPU, with a memory footprint at 145% of HBM capacity. Contiguous migration regions reflect spatial locality, and phase-dependent migration activity indicates differences in temporal reuse. Because these execution phases exhibit recurring kernel-launch patterns across iterations, kernel-launch progress provides a coarse-grained indication of changing memory-access behavior without tracking individual accesses. Spatial locality favors bulk transfers, while temporal reuse affects whether migration costs can be amortized before eviction; exploiting both requires balancing HBM residency against repeated migration and direct CPU-memory access.

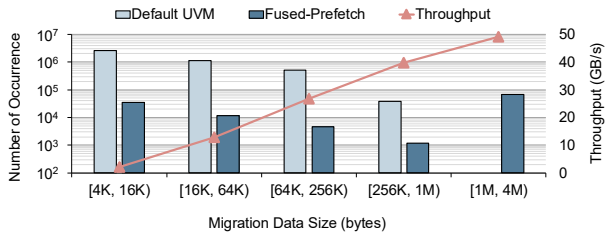

Figure 4: Number of occurrences and throughput of data migration events by migration size for UVM and Fused-Prefetch during Llama3-8B training on an H100 GPU.

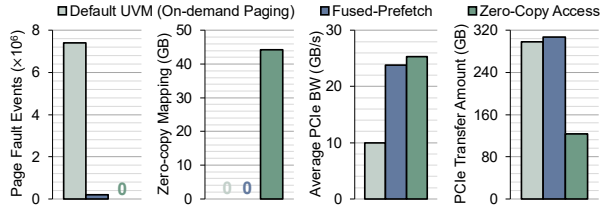

Figure 5: UVM and PCIe statistics of UVM, Fused-Prefetch, and Zero-Copy Access during Llama3-8B training on an H100 GPU with memory oversubscription. Page faults and PCIe transfer volume are measured over a single training iteration.

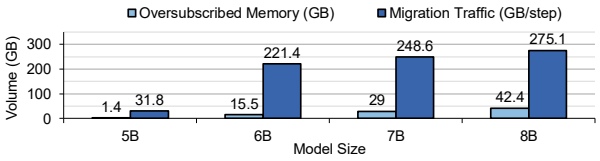

Figure 6: GPU memory oversubscription and migration traffic per training step for different Llama3-8B variant sizes.

3 Characterizing Memory Oversubscription

We present three key observations from characterizing driver-runtime-managed GPU memory oversubscription during LLM training: the limits of migration-efficiency optimization, the amplification of data movement under memory pressure, and the sensitivity of performance to zero-copy activation.

3.1 Transfer Efficiency vs. Data Movement

(01) **Optimizing Fine-Grained Migration Is Insufficient.** The repetitive execution and memory-access patterns of LLM training create opportunities to anticipate data accesses and proactively manage CPU–GPU transfers. We observe that conventional driver-runtime-managed oversubscription is dominated by small host-to-device migrations, typically ranging from a few KB to tens of KB, during Llama3-8B training on a single NVIDIA H100 GPU at 145% memory oversubscription (Figure 4). Such transfers achieve low effective PCIe bandwidth because per-migration overheads are poorly amortized.

In contrast, device-to-host evictions occur in 2 MB units and achieve higher effective bandwidth. Exploiting the spatial locality of these accesses, we fuse small migration requests into larger asynchronous prefetch operations, termed *Fused-Prefetch*, to improve transfer efficiency. This shifts the migration-size distribution toward transfers exceeding 1 MB (Figure 4), reducing page faults by 97% and improving average PCIe bandwidth utilization by 2.4× compared with conventional demand-driven migration (Figure 5). Although fused prefetching improves effective PCIe bandwidth, it still relies on migrating CPU-resident pages into HBM rather than accessing them directly. Thus, improving transfer efficiency does not necessarily reduce the total volume of data migrated or the overhead of maintaining an oversubscribed HBM working set.

3.2 Migration vs. Direct Access

(02) **Faster Migration Does Not Prevent Thrashing.** Under HBM oversubscription, migrating CPU-resident pages into GPU memory can require evicting existing pages, causing working sets to compete for limited HBM capacity. Subsequent reuse of evicted pages can trigger further migrations, moving the same data repeatedly between CPU DRAM and HBM. Even modest oversubscription can therefore induce substantial data movement. Figure 6 shows Llama3-8B variants with 5B–8B parameters trained on a single H100 GPU with 94 GB of HBM: although they exceed HBM capacity by only 1.4–42.4 GB, demand-driven migration generates bidirectional traffic equivalent to 6.5–22.7× the excess memory footprint per training step. For example, the 6B variant exceeds HBM capacity by just 15.5 GB but generates 221.4 GB of migration traffic per step, a 14.3× amplification. This behavior is particularly inefficient for training state with long reuse distances, where migration costs may not be amortized before eviction.

Zero-copy provides an alternative access path: the GPU accesses CPU-resident pages directly over PCIe without first migrating them into HBM, avoiding HBM capacity consumption for those pages and the associated migration and eviction overheads, but incurring remote-memory access costs. For Llama3-8B training at 145% memory oversubscription, zero-copy reduces measured PCIe transfer volume by approximately 3× and improves effective bandwidth utilization by 2.5× compared with conventional demand-driven migration (Figure 5). Although zero-copy accesses still generate PCIe traffic, these results demonstrate that avoiding migration can reduce data movement beyond what more efficient migration alone can achieve. Its benefit, however, depends on data reuse: frequently reused data benefits from HBM residency, whereas data with limited reuse may be more efficiently accessed directly from CPU memory.

3.3 Execution Progress and Memory Access

(03) **Naive Zero-Copy Degrades Performance.** LLM training exhibits heterogeneous memory-access patterns: frequently accessed data, such as activations and half-precision model parameters, benefits from HBM residency, whereas data with longer reuse intervals, including optimizer states and full-precision parameter copies, can be served directly from CPU DRAM. Forward propagation, backward propagation, and optimizer updates recur every iteration, and we observe recurring memory-migration patterns across iterations (Figure 3), suggesting that execution progress can indicate changes in memory-access behavior. To evaluate the sensitivity to activation

timing, we examine how training throughput varies when zero-copy is enabled at different points within an iteration, expressed as a fraction of completed kernel launches (§5.5). Enabling zero-copy too early can prevent frequently accessed pages from establishing HBM residency, whereas enabling it too late can incur unnecessary migration and eviction overhead. Poorly timed activation can degrade throughput by more than 50×, whereas activation within a broad window spanning approximately 40–90% of kernel launches achieves comparable performance in the evaluated configuration.

This broad window suggests that recurring kernel-launch progress can provide a practical signal for guiding zero-copy activation without reconstructing tensor lifetimes, predicting page accesses, or explicitly classifying training state. Execution progress does not identify which pages reside in HBM or predict their future reuse; instead, it lets access-path selection align with existing driver-managed memory residency.

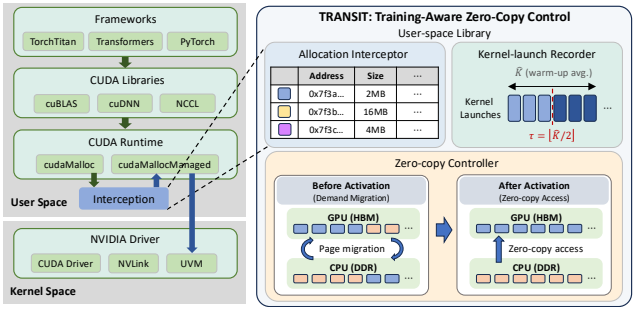

Figure 7: TRANSIT architecture: a user-space allocation interceptor and activity recorder provide the managed-memory regions and execution progress used to coordinate access through CUDA runtime interfaces.

4 TRANSIT: Overview

Building on the observations in §3, efficient scale-in requires GPU memory oversubscription to preserve per-GPU training throughput on fewer GPUs; otherwise, savings in GPU allocation are offset by longer training times. We identify four requirements. (R1) *Efficient data movement*: the system should preserve HBM residency for frequently accessed data while avoiding migrations whose costs cannot be amortized by subsequent accesses. (R2) *Low overhead*: memory-access-path selection should not require tensor-level memory management, page-access tracking, or significant runtime overhead. (R3) *Multi-node effectiveness*: benefits should hold across distributed training configurations, where overhead on individual GPUs can delay synchronized training steps, without introducing additional communication or synchronization. (R4) *Transparency*: the system should require no modifications to training applications, frameworks, communication libraries, schedulers, or device drivers.

4.1 System Architecture

Figure 7 illustrates TRANSIT's architecture, which consists of three components within a user-space shared library, each maintained independently by every GPU worker process. The *memory allocation interceptor* redirects supported GPU allocations to driver-runtime-managed memory (*e.g.,* CUDA Unified Virtual Memory [7] or HIP Managed Memory [1]) and records each allocation's base address and size in a process-local region table, without requiring tensor semantics or determining where individual pages reside. The *kernel-launch activity recorder* identifies training-iteration boundaries and counts kernel launches per iteration during initial training iterations, providing a coarse-grained measure of execution progress at the cost of a counter update per launch. The *zero-copy controller* uses these counts to set an activation threshold and, once it is reached, enables direct GPU access to CPU-resident pages in the recorded regions; its cost scales with the number of recorded regions rather than the number of pages. TRANSIT thus proceeds in three stages: initialization, observation, and activation.

Transparent Memory Oversubscription. Managed allocations share a unified virtual address space accessible by the GPU, allowing training state to exceed HBM capacity without application changes. Before activation, the GPU driver and runtime manage residency unchanged: a GPU access to a nonresident page raises a page fault that the driver services by migrating the page from CPU DRAM into HBM, stalling the faulting access until migration completes. Under memory pressure, the driver may evict resident pages to CPU DRAM (§3.2). During initial training execution, frequently accessed pages, such as activations and half-precision parameters, therefore become resident in HBM. In contrast, pages with longer reuse intervals, such as optimizer states and full-precision parameter copies, may be evicted as the working set competes for limited HBM capacity. TRANSIT preserves this driver-managed residency and only changes how the GPU subsequently accesses CPU-resident pages, reducing migration pressure without explicitly selecting which pages reside in HBM or CPU memory.

Execution-Guided Activation. Algorithm 1 summarizes the procedure, which measures kernel-launch counts during warm-up (Phase 1) and enables zero-copy access at the resulting threshold (Phase 2). As established in §3.3, LLM training exhibits recurring kernel-execution patterns across iterations. During the first W iterations, the recorder counts the kernel launches K_i in each iteration i, identifying iteration boundaries by intercepting `optimizer.step()` and observing launches through GPU runtime profiling callbacks, and computes $\widehat{K} = \frac{1}{W}\sum_{i=1}^{W} K_i$. We use $W = 2$ across all evaluated workloads; these iterations both characterize execution progress and allow the memory manager to establish an initial HBM working set. The activation threshold is $\tau = \lfloor \widehat{K}/2 \rfloor$, the iteration midpoint, which lies within the broad favorable window identified in §3.3. Rather than identifying an exact transition between training phases, τ provides a workload-relative execution point determined through lightweight kernel-launch monitoring, without tracking individual memory accesses or reconstructing tensor lifetimes. We use launch counts rather than elapsed time because an iteration's kernel sequence is repeatable, whereas iteration time varies with communication delays and stragglers. Because kernel launches are asynchronous, host-side launch progress may run ahead of device execution; the breadth of the favorable window tolerates this offset. Each worker computes its own τ, so ranks that execute different kernel sequences, such as pipeline stages, require no coordination.

Algorithm 1 TRANSIT: Execution-Guided Zero-Copy Activation

Require: GPU d, warm-up iterations W, managed-memory regions $\mathcal{R}$

$K_{\text{sum}} \leftarrow 0$
Phase 1: Measure kernel-launch counts
for $i = 1$ to W **do**
  $k \leftarrow 0$
  for each kernel launch in iteration i **do**
    $k \leftarrow k + 1$
  end for
  $K_{\text{sum}} \leftarrow K_{\text{sum}} + k$
end for
$\widehat{K} \leftarrow K_{\text{sum}}/W$
$\tau \leftarrow \lfloor \widehat{K}/2 \rfloor$
Phase 2: Enable zero-copy access
$k \leftarrow 0$
for each kernel launch in the next iteration **do**
  $k \leftarrow k + 1$
  if $k \geq \tau$ **then**
    for each region $(p, n) \in \mathcal{R}$ **do**
      EnableZeroCopy(p, n, d)
    end for
    break
  end if
end for

In the next iteration, demand-driven migration continues until the launch count reaches τ, allowing frequently accessed data to establish HBM residency before the access-path transition. At that point, the controller enables *zero-copy access* to every recorded region. Zero-copy establishes GPU page-table mappings to CPU-resident pages without migrating their contents or allocating HBM. GPU loads and stores to these pages are then issued as fine-grained memory transactions over PCIe and served through the GPU's L2 cache (Figure 2), rather than raising page faults that migrate entire pages into HBM. Pages already resident in HBM remain locally mapped and continue to be accessed at HBM bandwidth. After activation, CPU-resident pages no longer fault and are therefore not migrated into HBM, so steady-state residency is largely fixed at the activation point; the driver continues to manage eviction. Zero-copy thus trades a per-access interconnect cost for eliminating migration, eviction, and fault handling for CPU-resident data, addressing the tradeoff identified in §3.2.

Steady-state memory access. The mappings are established once and remain effective throughout subsequent training; TRANSIT neither switches back to migration nor recomputes τ, and kernel-launch monitoring is no longer required for access-path selection. Activation thus preserves the page residency, and τ determines its timing: by mid-iteration, the reused working set (parameters, gradients, and activations) is HBM-resident while once-per-step optimizer state has been evicted, and because iterations repeat, this residency state remains valid across subsequent iterations. TRANSIT never identifies hot or cold pages or enforces their placement: the resulting behavior emerges from the memory manager's residency decisions before activation and the direct-access mappings established afterward.

Memory consistency. TRANSIT changes only how the GPU maps managed pages: enabling direct access neither copies nor migrates data, and the underlying memory manager remains responsible for keeping mappings valid. TRANSIT therefore inherits the memory-consistency semantics of driver-runtime-managed memory, introduces no separate coherence protocol, and does not alter the application's synchronization requirements; concurrent accesses to shared data follow the platform's standard synchronization and memory-ordering rules.

4.2 Implementation and Deployment

We implement TRANSIT for NVIDIA GPUs as a user-space shared library that maps each design component onto CUDA Unified Virtual Memory (UVM) and the CUDA Profiling Tools Interface (CUPTI), requiring no modifications to the training stack.

TRANSIT is loaded via `LD_PRELOAD` and interposes on CUDA runtime allocation entry points (*e.g.,* `cudaMalloc`), forwarding all other calls to the original library. Each intercepted request is served by `cudaMallocManaged`, which reserves a range in the unified virtual address space without immediately backing it with HBM; physical pages are populated on first touch and thereafter migrated or evicted by the UVM driver at page granularity. Because frameworks such as PyTorch obtain device memory through a caching allocator that requests large segments and sub-allocates tensors within them, interposing at this level covers training state without framework changes and keeps the region table small. The interceptor records each region's base address and size in a process-local table. Oversubscription requires a GPU and driver that support demand paging of managed memory, as on the evaluated H100 systems.

The recorder subscribes to CUPTI runtime and driver API callbacks for kernel-launch entry points and increments a process-local counter on each launch; it records no arguments, timestamps, or memory addresses, so the per-launch cost is a callback and a counter update. Iteration boundaries are identified by intercepting Python-level `optimizer.step()` calls through a runtime-installed hook, without modifying the training application, and the kernel-launch counter K_i is recorded and reset at each boundary. Kernels launched as part of a CUDA Graph are not observed individually; the evaluated configurations do not use CUDA Graphs, and supporting them would require counting graph launches or the kernels within them.

When the counter reaches τ, the controller invokes `cudaMemAdvise` with `cudaMemAdviseSetAccessedBy` on each recorded region, specifying the local GPU. This advice directs the driver to keep the region mapped in the GPU's page tables regardless of where its pages reside: CPU-resident pages become directly accessible over PCIe without faulting, HBM-resident pages remain local, and if the driver later evicts a page to CPU memory, the GPU mapping is updated rather than removed, so subsequent accesses are served remotely instead of triggering re-migration [3]. The controller issues no explicit migration, prefetch, or eviction commands, leaving residency decisions to UVM; activation costs one advice call per region, independent of the number of pages. `cudaMemAdviseSetAccessedBy` establishes mappings only; unlike `cudaMemAdviseSetReadMostly`, it creates no read-only replicas, so managed-memory consistency semantics are unchanged. HIP exposes analogous interfaces (`hipMallocManaged` and

hipMemAdviseSetAccessedBy), so the design could be extended to AMD GPUs; we leave this port to future work.

TRANSIT runs in multi-GPU and multi-node environments, including container-orchestrated clusters using Slurm, Kubernetes, and Red Hat OpenShift. Deployment requires only making the shared library available inside the training container and setting LD_PRELOAD at job launch; evaluated configurations need no privileged containers, kernel modules, or modifications to the host operating system, GPU driver, or cluster scheduler. Because all TRANSIT state is process-local, FSDP, TP, and PP retain their original communication and synchronization behavior, and NCCL runs unmodified.

5 Evaluation

We evaluate TRANSIT to answer four questions: (1) How effectively does training-aware zero-copy access mitigate the overhead of GPU memory oversubscription compared with existing offloading approaches? (2) How much can GPU allocation be reduced while preserving per-GPU training throughput? (3) How does scale-in affect performance under constrained inter-node bandwidth? (4) How do smaller GPU allocations translate into cluster-level scheduling and communication benefits?

5.1 Experimental Setup

Training workload. We use TorchTitan [26] as the training framework and evaluate four representative models: Llama3-70B [18], Llama4-Scout [28], Qwen3-32B [48], and Qwen3-30B-A3B [48], as summarized in Table 1. Our evaluation covers fully sharded data parallelism (FSDP) and hybrid configurations that combine intra-node tensor parallelism (TP) with inter-node FSDP and pipeline parallelism (PP). We use a local batch size of 1 for dense models and 4 for MoE models, increasing it for PP to mitigate pipeline bubbles. To ensure compatibility across GPU counts with different data-parallel degrees, we set the global batch size to their common multiple. We use TorchTitan's default mixed-precision training (FP32 parameters and BF16 computation), except for TP+PP configurations, which use BF16 throughout due to TorchTitan's limitations. Full activation checkpointing is enabled across all configurations.

Metrics and baselines. We report per-GPU TFLOPS, computed as floating-point operations per iteration divided by iteration time and GPU count; aggregate throughput is the product of per-GPU TFLOPS and GPU count. We measure per-node network traffic using NVIDIA ConnectX RoCE hardware counters exposed through Linux sysfs, summing transmitted and received bytes per iteration. We also report the CPU-resident memory footprint and PCIe traffic to characterize offloading overhead.

For each workload, the no-offloading baseline uses the minimum GPU count required without offloading. At reduced GPU counts, we compare TRANSIT against default UVM, Fused-Prefetch (§3.1), TorchTitan-Offload, ZeRO-Offload [38], and ZeRO-Infinity [37]. UVM uses default managed-memory behavior, while Fused-Prefetch coalesces prefetch transfers. TorchTitan-Offload uses enable_cpu_offloading=true, while ZeRO-Offload and ZeRO-Infinity use their corresponding DeepSpeed configurations.

Multi-node cluster. We evaluate TRANSIT on two H100-based GPU clusters deployed in OpenShift, as summarized in Table 2. Cluster-A spans two racks with eight nodes each, interconnected through directly connected top-of-rack (ToR) switches providing 10.4 Tbps of aggregate switching capacity. Unless otherwise specified, all NICs operate at their nominal 400 Gbps line rate.

We additionally evaluate bandwidth-limited configurations to capture settings where the available network bandwidth is lower than the nominal line rate. Large-scale GPU clusters are increasingly exposed to partial hardware and network degradations, where failures or degraded components can reduce the throughput of otherwise healthy distributed training jobs [8, 24]. Beyond these degradations, bandwidth limitations also arise from deployment constraints, as academic or shared research clusters may not be provisioned with uniformly high-bandwidth interconnects. To study such cases in a controlled setting, we emulate a degraded or bandwidth-limited scenario by capping the NICs to 100 Gbps using mlnx_qos.

Table 1: Summary of the evaluated models, their architectural types, parallelization strategies, global and local batch sizes (GB and LB), and the corresponding minimum GPU count required without memory offloading.

Model	Type	Parallelism	GB	LB	Min. Num. of GPUs
Qwen3-32B	Dense	FSDP	128	1	8
Llama3-70B	Dense	FSDP	240	1	24
		FSDP+TP	48	1	16
		TP+PP	30	30	12
Qwen3-30B-A3B	MoE	FSDP	960	4	20
Llama4-Scout	MoE	FSDP+TP	64	4	64

Table 2: Summary of the multi-node cluster configurations used for the evaluation.

Configuration	Cluster-A	Cluster-B
Nodes	16	3
GPUs per node	4× H100 (80GB)	8× H100 (80GB)
CPU–GPU interconnect	PCIe Gen4	PCIe Gen5
NICs per node	4× ConnectX-7	8× ConnectX-7
NIC line rate	400 Gbps	400 Gbps
NVLink BW per GPU	300 GB/s	900 GB/s
Host CPU	Intel Sapphire Rapids, 80 cores	
CPU memory per node	2 TB	

5.2 Memory Oversubscription Performance

Impact of memory access strategies. Figure 8 compares Llama3-70B FSDP training on Cluster-A as the allocation shrinks below the 24-GPU, six-node no-offloading baseline. At the same reduced GPU counts, both Fused-Prefetch and TRANSIT outperform UVM, improving per-GPU TFLOPS by 4–99% and 10–243%, respectively. While Fused-Prefetch improves transfer efficiency, it retains migration and eviction overheads under HBM pressure, whereas TRANSIT mitigates migration thrashing through zero-copy access. Notably, with 17–33% fewer GPUs, TRANSIT slightly exceeds the no-offloading baseline by 3–4%, as reduced communication volume outweighs CPU-memory access overhead. As the GPU count decreases further, each GPU holds a larger shard of the training state, increasing CPU-memory accesses and reducing per-GPU throughput. Nevertheless,

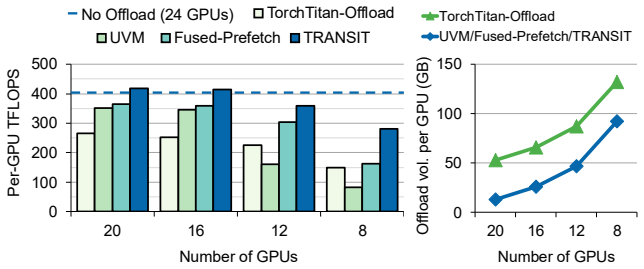

Figure 8: Per-GPU training throughput and CPU-resident memory footprint across GPU counts for Llama3-70B training with FSDP. The dashed line on the left panel indicates the 24-GPU no-offloading baseline.

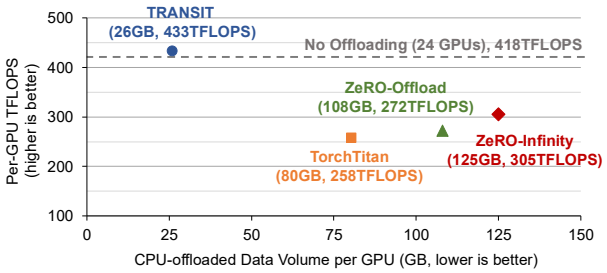

Figure 9: Per-GPU training throughput and offloaded data volume of `TRANSIT` and framework-managed offloading approaches for Llama3-70B training on 16 GPUs. The dashed horizontal line indicates the 24-GPU no-offloading baseline.

`TRANSIT` consistently outperforms TorchTitan-Offload, maintaining 57–87% higher per-GPU TFLOPS at aggressive scale-in, where UVM and Fused-Prefetch fall below TorchTitan-Offload. As shown in the right panel of Figure 8, all three UVM-based approaches maintain substantially smaller CPU-resident memory footprints than TorchTitan-Offload, as they spill only the memory exceeding HBM capacity rather than entire classes of training state.

Framework-managed offloading. Figure 9 compares offloading mechanisms for Llama3-70B on 16 GPUs across two Cluster-B nodes. The no-offloading baseline requires 24 GPUs and achieves 418 TFLOPS. At 16 GPUs, `TRANSIT` achieves 433 TFLOPS, surpassing the baseline in per-GPU efficiency as reduced inter-node communication outweighs the overhead of CPU-memory accesses. In contrast, TorchTitan-Offload, ZeRO-Offload, and ZeRO-Infinity achieve 258, 272, and 305 TFLOPS, respectively. `TRANSIT` thus delivers 68%, 59%, and 42% higher per-GPU throughput than these framework-managed approaches, respectively. Two differences in memory placement and access account for the observed performance gap. First, these baselines offload entire classes of training state regardless of how much exceeds HBM capacity, placing 80, 108, and 125 GB per GPU in CPU memory, whereas `TRANSIT` spills only the overflow (26 GB, or 3.1–4.8× less) because driver-managed placement fills HBM before spilling. Because these frameworks expose offloading only at the granularity of state classes, users cannot size the spill to the actual overflow; `TRANSIT` does so automatically, as the overflow

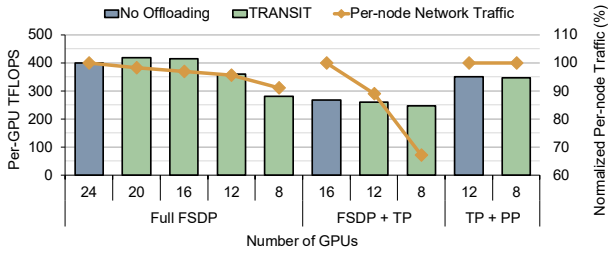

Figure 10: Per-GPU TFLOPS and normalized per-node network traffic for Llama3-70B training across parallelism strategies and GPU counts.

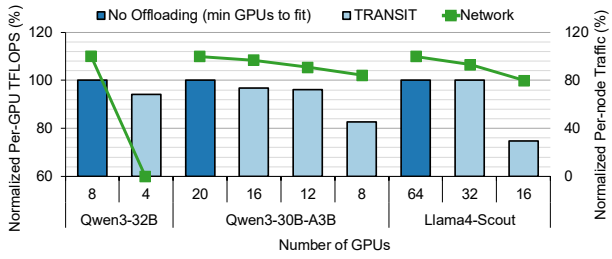

Figure 11: Per-GPU TFLOPS of the no-offloading baseline and `TRANSIT` under scale-in across models, along with per-node network traffic normalized to the baseline.

changes with model, parallelism, and GPU count. Second, `TRANSIT` accesses spilled state in place via zero-copy, whereas ZeRO-Offload and ZeRO-Infinity move it to host CPU cores for the optimizer update; a small spill alone is insufficient, as UVM and Fused-Prefetch spill the same volume yet underperform `TRANSIT` (Figure 8).

5.3 Scale-In Efficiency

Different parallelisms. Figure 10 shows the scaling behavior of Llama3-70B training on Cluster-A across different parallelization strategies (FSDP, FSDP+TP, and TP+PP), along with the corresponding per-node communication traffic as the number of nodes decreases. Across all parallelization strategies, `TRANSIT` maintains over 90% of the per-GPU throughput at up to half the number of GPUs, relative to the no-offloading baseline. For FSDP and FSDP+TP, scale-in also reduces per-node communication volume per step. This trend arises because FSDP's collective operations (all-gather and reduce-scatter) scale with model sharding, causing communication volume to decrease as the number of participating devices is reduced. For full FSDP, the reduction is gradual since sharding occurs across all GPUs. In contrast, FSDP+TP exhibits a steeper reduction because FSDP operates across nodes, leading to a 33% decrease in per-node communication volume when scaling from 4 to 2 nodes. In contrast, TP+PP does not exhibit a similar reduction, as its inter-node communication is dominated by pipeline-parallel activation exchanges, which are point-to-point and remain largely invariant to the number of nodes.

Different models and scales. Figure 11 illustrates the scaling behavior of different models across varying node counts on Cluster-A.

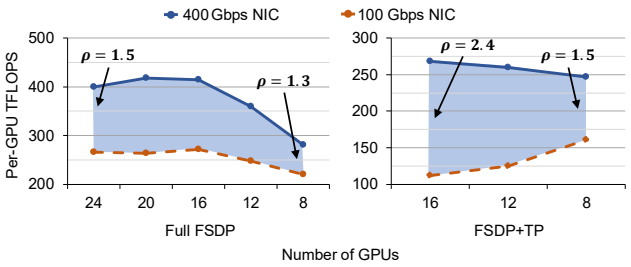

Figure 12: Per-GPU TFLOPS for Llama3-70B training across parallelism strategies and GPU counts under different NIC bandwidths (400 Gbps vs. 100 Gbps), along with the corresponding bandwidth sensitivity metric ρ.

Despite differences in model size and minimum GPU requirements, a consistent trend emerges: TRANSIT sustains per-GPU throughput comparable to the no-offloading baseline while using only half the number of GPUs. These results show that TRANSIT preserves high per-GPU efficiency across the evaluated dense and MoE models despite differences in memory footprint and parallelization. However, per-GPU TFLOPS decrease noticeably under aggressive scale-in. For example, throughput drops by 18% and 25% for Qwen3-30B-A3B and Llama4-Scout, respectively, when the GPU count is reduced by 60% and 75%. This degradation arises from higher memory oversubscription, which amplifies data movement overhead and limits the effectiveness of memory offloading.

Implications for training efficiency. Scale-in trades aggregate training throughput for a smaller GPU allocation. For a fixed training workload, aggregate throughput scales with the product of GPU count and per-GPU throughput, while GPU-hours are inversely proportional to per-GPU throughput. Thus, preserving per-GPU throughput limits the increase in GPU-hours despite longer training times on fewer GPUs. When per-GPU throughput exceeds the baseline, scale-in can additionally reduce total GPU-hours. These results highlight that TRANSIT primarily enables memory-efficient GPU consolidation, with further resource savings when reduced communication overhead offsets CPU-memory access costs.

5.4 Network-Constrained Training

Bandwidth sensitivity. We quantify bandwidth sensitivity as $\rho = P(400)/P(100)$, where $P(B)$ is training throughput at per-NIC bandwidth B (Gbps) for a fixed workload and GPU count. A ratio closer to one indicates lower sensitivity to network bandwidth.

Figure 12 shows that scale-in with TRANSIT reduces ρ from 1.5 to 1.3 when scaling FSDP from 24 to 8 GPUs, and from 2.4 to 1.5 when scaling FSDP+TP from 16 to 8 GPUs. Scale-in increases CPU-memory access overhead but reduces the communication component of iteration time, lowering the relative throughput penalty when NIC bandwidth drops from 400 to 100 Gbps. Notably, at 100 Gbps, eight-GPU TRANSIT FSDP+TP achieves 161 per-GPU TFLOPS, 44% higher than the 16-GPU no-offloading baseline (112 TFLOPS). This demonstrates that reduced communication overhead can outweigh the additional CPU-memory access cost, enabling higher per-GPU efficiency with half the GPUs.

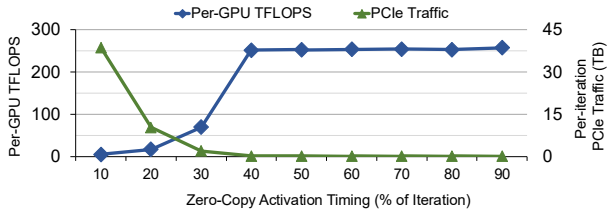

Figure 13: Sensitivity of Llama3-70B training throughput and per-iteration PCIe traffic to the zero-copy activation point.

5.5 Sensitivity Study

Zero-copy activation timing. We evaluate the sensitivity of zero-copy activation timing using a representative Llama3-70B FSDP training rank with the model states and memory footprint corresponding to an eight-GPU configuration. We vary the activation point across an iteration, expressed as the fraction of completed kernel launches, and measure the resulting per-GPU training throughput. Figure 13 shows that early zero-copy activation severely degrades performance. At 10%, throughput drops to 4.8 TFLOPS, with 38.6 TB of PCIe traffic per iteration. This occurs because optimizer updates in the preceding iteration can displace frequently accessed data, such as activations, gradients, and half-precision parameters, from HBM. Enabling zero-copy too early prevents UVM from restoring their HBM residency, resulting in excessive remote accesses. Delaying activation allows UVM to restore hot data to HBM while evicting less frequently accessed optimizer states to CPU memory. Beyond 40%, throughput remains stable at 251–257 TFLOPS, with PCIe traffic below 0.23 TB per iteration. The broad favorable window supports using kernel-launch progress as a low-overhead activation heuristic, without identifying individual memory accesses or training phases.

5.6 Cluster-level Simulation

Simulator design. We build a discrete-event simulator using production traces to evaluate the cluster-wide impact of TRANSIT. The simulator models multi-rack clusters with 4 or 8 GPUs per node and 32 GPUs per rack, varying the number of racks to scale cluster capacity. The scheduler prioritizes intra-rack placement using packing or spreading strategies [6, 42], falling back to multi-rack placement when necessary.

For cluster-level evaluation, we compare two scenarios: (1) a baseline that replays the original trace without modification and (2) a TRANSIT-enabled scenario that reduces node allocations by 25–50% for eligible jobs. We leave single-node jobs unchanged and apply 25–50% node reductions for multi-node jobs, representing workloads whose memory-driven allocations permit scale-in. Assuming unchanged per-GPU throughput, job duration initially scales inversely with GPU allocation (*e.g.*, halving the allocation doubles duration); placement-dependent throughput adjustments are then applied. On top of this, the simulator distinguishes cases where cross-rack communication is eliminated, introduced, or partially reduced, and incorporates these effects into job duration. The simulator incorporates scaling factors to capture these effects under different placement transitions, extracted from the results in §5.4.

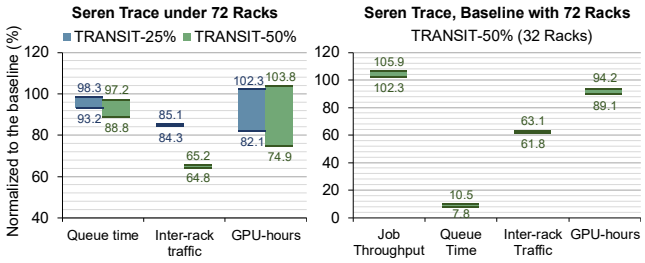

Figure 14: Cluster-level impact of TRANSIT on the Seren trace: (a) full-scale 72-rack cluster and (b) reduced-scale 32-rack TRANSIT deployment compared against the 72-rack baseline. Bars show normalized results with ranges.

Specifically, when an inter-rack job becomes intra-rack, we apply a speedup factor derived from TRANSIT at 400 Gbps NIC configuration, normalized to a no-offloading baseline at 100 Gbps NIC configuration (0–137%). When an intra-rack job becomes inter-rack, we apply a slowdown factor based on TRANSIT at 100 Gbps NIC configuration, normalized to a 400 Gbps baseline NIC configuration (0–55%). For jobs that remain intra-rack or inter-rack before and after applying TRANSIT, we apply a scaling factor corresponding to TRANSIT performance relative to the baseline under the same placement and network setting (ranging from −5% to 5%). The simulator reports a range of cluster-level metrics across both eligible jobs and all jobs, including cluster throughput, GPU-hours per job, queue time, and inter-rack traffic.

Workload. Input traces are preprocessed from the ACME datacenter trace [21], specifically the Kalos and Seren subsets. Kalos represents a relatively lightly loaded workload, while Seren captures a more heavily loaded setting.

Performance improvement across different loads. We begin with the Seren trace on a full-scale 72-rack cluster, representing a heavily loaded setting. As shown in Figure 14a, TRANSIT reduces inter-rack traffic by 14.9–15.7% and 34.8–35.2% at 25% and 50% node count reductions, respectively, demonstrating that greater node consolidation consistently improves placement locality. This improved locality also translates into system-level benefits, reducing average eligible-job queue time by 1.7–6.8% and 2.8–11.2%, respectively. Moreover, TRANSIT reduces average GPU-hours per job by up to 18.0% and 25.1% at 25% and 50% node count reductions, respectively. Only under the worst-case combination of slowdowns from node count reduction and unfavorable inter-rack placement transitions do GPU-hours increase, by up to 2.3% and 3.8%, respectively. In the 8 GPUs per node configuration, 40.6% of eligible jobs collapse to a single node under 50% node count reduction, eliminating inter-node communication for those jobs.

We next evaluate the Kalos trace on a full 76-rack cluster, representing a lightly loaded setting. Here, cluster-level queue time benefits do not materialize because freed nodes remain idle rather than serving queued jobs. Nevertheless, inter-rack traffic decreases by 24.2% and 49.2% under 25% and 50% node count reductions, respectively, while idle GPU time changes by less than 1%.

Performance under reduced cluster size. We next evaluate TRANSIT under capacity constraints using the Seren trace on a reduced 32-rack cluster, less than half the 72-rack production deployment, and compare against a baseline running the same workload on the full 72-rack cluster without TRANSIT. As shown in Section 5.6, at 50% node count reduction, TRANSIT improves eligible-job throughput by 2.3–5.9%, reduces average eligible-job queue time by 89.5–92.2%, and decreases inter-rack traffic by 36.9–38.2% across the evaluated throughput assumptions. Furthermore, GPU-hours per job decrease by 5.8–10.9%, showing that, under the modeled scale-in and placement assumptions, smaller GPU allocations can offset substantially reduced cluster capacity.

For the Kalos trace on a reduced 32-rack cluster, TRANSIT reduces queue time by 85.2% and 95.3% at 25% and 50% node count reductions, respectively, relative to the full-scale baseline. Notably, varying the assumed throughput improvement for jobs transitioning from inter-rack to intra-rack communication has negligible impact, indicating that the gains are driven primarily by node count reduction and improved placement locality rather than per-job throughput changes.

6 Related Work

Training LLMs with data offloading. Fine-grained systems such as G10 [50], TeraIO [49], and SSDTrain [45] exploit tensor lifetimes and execution characteristics to schedule data movement across GPU HBM, CPU DRAM, and NVMe SSDs. While effective, they require tight framework integration, which becomes increasingly complex under multi-dimensional parallelism. In contrast, ZeRO-Offload [38], ZeRO-Infinity [37], and TorchTitan [26] favor simpler, coarse-grained offloading of entire data classes (*e.g.,* parameters or optimizer states). Application-transparent offloading via NVIDIA Unified Virtual Memory (UVM) [7] decouples memory management from training frameworks. However, on-demand migration can incur repeated page faults under memory pressure, motivating custom prefetching and placement policies [23, 27]. Forest [27] modifies the UVM driver to control page placement and migration, while DeepUM [23] introduces kernel-space components and runtime integration to intercept page faults and orchestrate prefetching. More broadly, G10, TeraIO, DeepUM, and Forest exploit tensor lifetimes or kernel-access correlations to guide data movement and placement. In contrast, TRANSIT uses recurring kernel-execution patterns to guide zero-copy access while leaving page residency to the driver, without requiring kernel modifications.

Communication optimizations for distributed LLM training. Prior work reduces communication overhead through compression, parallelism, topology-aware placement, and computation–communication overlap. ZeRO++ [43] reduces communication volume through low-precision transfers and hierarchical partitioning. Pipeline parallelism replaces global synchronization with stage-wise communication, while improved schedules mitigate pipeline inefficiencies [20, 31]. Prior work improves distributed training efficiency through communication-aware scheduling and resource allocation [10, 19, 32, 52], while ByteScheduler [33] overlaps communication with computation through priority-based scheduling. These approaches optimize communication within a given GPU allocation but do not directly address GPU allocations driven by

training-state memory requirements. In contrast, TRANSIT reduces memory-driven GPU requirements, potentially reducing inter-node communication and network sensitivity.

7 Conclusion

Conventional distributed LLM training often scales out to meet GPU memory requirements, while memory oversubscription generally incurs substantial performance penalties. Our work challenges this tradeoff. We presented TRANSIT, demonstrating that execution-guided zero-copy access can make driver-managed memory oversubscription efficient without explicit data placement, prefetching, or framework modifications. By relaxing the coupling between GPU allocation and memory capacity requirements, TRANSIT enables aggressive scale-in while preserving high per-GPU computational efficiency. Our evaluation further shows that using fewer GPUs can reduce inter-node communication and, in production-trace-driven simulation, improve cluster-level scheduling efficiency. Together, these results show that memory oversubscription can be a practical means of improving distributed LLM training efficiency, rather than merely serving as a fallback for insufficient GPU memory.

Acknowledgement

This work was supported in part by grants from the IBM-Illinois Discovery Accelerator Institute (IIDAI) and the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. RS-2025-02214497, Development of low-level optimization program API technology for AI semiconductors). The authors acknowledge the National Artificial Intelligence Research Resource (NAIRR) Pilot (NAIRR250426), the AI Alliance and resources from IBM, Red Hat and the Mass Open Cloud for contributing to this research result. The authors thank Padmanabhan Santhanam (IBM Research) for his constant support during the NAIRR Pilot administrative process, and Kevin O'Connor (IBM Reserach), Jason Schlessman (Red Hat), Taj Salawu (Red Hat), Naved Ansari (MOC), and Hakan Saplakoglu (MOC), for their support in provisioning and maintaining the infrastructure used for large-scale GPU experiments.

References

[1] Accessed in 2026. AMD ROCm Core SDK 10.1.0. https://rocm.docs.amd.com/projects/HIP/en/latest/doxygen/html/group___memory_m.html.

[2] Accessed in 2026. ConnectX NICs. https://www.nvidia.com/en-us/networking/ethernet-adapters/.

[3] Accessed in 2026. CUDA Programming Guide v13.5. https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/unified-memory.html

[4] Accessed in 2026. CUDA semantics. https://docs.pytorch.org/docs/2.12/notes/cuda.html.

[5] Accessed in 2026. Developing a Linux Kernel Module using GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[6] Accessed in 2026. Pod Topology Spread Constraints. https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/.

[7] Accessed in 2026. Unified Memory in CUDA 6. https://developer.nvidia.com/blog/unified-memory-in-cuda-6/.

[8] Daiyaan Arfeen, Dheevatsa Mudigere, Ankit More, Bhargava Gopireddy, Ahmet Inci, and Gregory R. Ganger. 2025. Nonuniform-Tensor-Parallelism: Mitigating GPU failure impact for Scaled-up LLM Training. arXiv:2504.06095 [cs.DC] https://arxiv.org/abs/2504.06095

[9] Sascha Brodsky. 2024. The Hidden Costs of AI: How Generative Models Are Reshaping Corporate Budgets. https://www.ibm.com/think/insights/ai-economics-compute-cost. Accessed: April 15, 2026.

[10] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. 2024. Crux: Gpu-efficient communication scheduling for deep learning training. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 1–15.

[11] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[12] Runxiang Cheng, Chris Cai, Selman Yilmaz, Rahul Mitra, Malay Bag, Mrinmoy Ghosh, and Tianyin Xu. 2023. Towards GPU memory efficiency for distributed training at scale. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. 281–297.

[13] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.

[14] Jared Fernandez, Luca Wehrstedt, Leonid Shamis, Mostafa Elhoushi, Kalyan Saladi, Yonatan Bisk, Emma Strubell, and Jacob Kahn. 2024. Hardware scaling trends and diminishing returns in large-scale distributed training. *arXiv preprint arXiv:2411.13055* (2024).

[15] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. 2024. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 57–70.

[16] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. 2024. Ai and memory wall. *IEEE Micro* 44, 3 (2024), 33–39.

[17] Soudeh Ghorbani, Yimeng Zhao, Srikanth Sundaresan, Ying Zhang, Yijing Zeng, Abhigyan Sharma, Prashanth Kannan, and Cristian Lumezanu. 2025. Congestion Patterns in a Large-scale RDMA Datacenter. In *Proceedings of the 2025 ACM Internet Measurement Conference*. 944–951.

[18] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu

Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[19] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 485–500. https://www.usenix.org/conference/nsdi19/presentation/gu

[20] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377* (2018).

[21] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. 2024. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 709–729.

[22] 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

[23] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. 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, Volume 2*. 207–221.

[24] Apostolos Kokolis, Michael Kuchnik, John Hoffman, Adithya Kumar, Parth Malani, Faye Ma, Zachary DeVito, Shubho Sengupta, Kalyan Saladi, and Carole-Jean Wu. 2025. Revisiting reliability in large-scale machine learning research clusters. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1259–1274.

[25] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. arXiv:2006.16668 [cs.CL] https://arxiv.org/abs/2006.16668

[26] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. 2025. TorchTitan: One-stop PyTorch native solution for production ready LLM pretraining. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=SFN6Wm7YBI

[27] Mao Lin, Yuan Feng, Guilherme Cox, and Hyeran Jeon. 2025. Forest: Access-aware GPU UVM Management. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 137–152.

[28] Meta AI. 2025. Llama 4 Model Card. https://github.com/meta-llama/llama-models/blob/main/models/llama4/MODEL_CARD.md.

[29] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *International Conference on Learning Representations*. https://openreview.net/forum?id=r1gs9JgRZ

[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. https://doi.org/10.1145/3341301.3359646

[31] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–15.

[32] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.

[33] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 16–29.

[34] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, et al. 2024. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 691–706.

[35] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. 2024. {CASSINI}:{Network-Aware} job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1403–1420.

[36] 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.

[37] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 59, 14 pages. https://doi.org/10.1145/3458817.3476205

[38] 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.

[39] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2023. {TACCL}: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 593–612.

[40] 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).

[41] Min Si, Pavan Balaji, Yongzhou Chen, Ching-Hsiang Chu, Adi Gangidi, Saif Hasan, Subodh Iyengar, Dan Johnson, Bingzhe Liu, Regina Ren, et al. 2025. Collective communication for 100k+ GPUs. *arXiv preprint arXiv:2510.20171* (2025).

[42] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the tenth european conference on computer systems*. 1–17.

[43] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Connor Holmes, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. 2023. Zero++: Extremely efficient collective communication for giant model training. *arXiv preprint arXiv:2306.10209* (2023).

[44] Zheng Wang, Anna Cai, Xinfeng Xie, Zaifeng Pan, Yue Guan, Weiwei Chu, Jie Wang, Shikai Li, Jianyu Huang, Chris Cai, et al. 2025. {WLB-LLM}:{Workload-Balanced} 4D Parallelism for Large Language Model Training. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 785–801.

[45] Kun Wu, Jeongmin Brian Park, Xiaofan Zhang, Mert Hidayetoğlu, Vikram Sharma Mailthody, Sitao Huang, Steve Lumetta, and Wen-Mei Hwu. 2025. SSDTrain: An activation offloading framework to ssds for faster large language model training. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–7.

[46] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. 2024. Falcon: Pinpointing and mitigating stragglers for large-scale hybrid-parallel training. *arXiv preprint arXiv:2410.12588* (2024).

[47] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 595–610.

[48] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[49] Ziqi Yuan, Haoyang Zhang, Yirui Zhou, Apoorve Mohan, I-Hsin Chung, Seetharami Seelam, and Jian Huang. 2025. Cost-Efficient LLM Training with Lifetime-Aware Tensor Offloading via GPUDirect Storage. In *Advances in Neural Information Processing Systems*, D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (Eds.), Vol. 38, Main Conference. Curran Associates, Inc., 49421–49444. https://doi.org/10.52202/085713-1653

[50] Haoyang Zhang, Yirui Zhou, Yuqi Xue, Yiqi Liu, and Jian Huang. 2023. G10: Enabling an efficient unified gpu memory and storage architecture with smart tensor migrations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 395–410.

[51] Xiaoyang Zhao, Zhe Zhang, and Chuan Wu. 2024. Adapcc: Making collective communication in distributed machine learning adaptive. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 25–35.

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