TurboBus: Pooling PCIe Bandwidth for LLM Workloads via Scale-Up Fabrics

Xinyu Yang, Kaiqiang Xu, Kai Chen
iSING Lab, Hong Kong University of Science and Technology

Abstract

GPU memory offloading is widely adopted for LLM workloads but shifts the bottleneck to GPU–CPU transfers, which can take up to 90% of the end-to-end inference/training time! Paradoxically, over 60% of PCIe bandwidth remains idle. The root cause is that PCIe links are *individually bottlenecked but collectively underutilized*. Bursty, phase-driven transfer patterns leave bandwidth idle both within and across jobs. We present TurboBus, which pools PCIe bandwidth across co-located jobs via emerging scale-up fabrics. TurboBus enables any GPU to borrow idle PCIe links from neighboring GPUs, even those belonging to other jobs, while preserving isolation through a privileged daemon. At the core of TurboBus, it streams data through relay GPUs with bounded memory overhead, keeps all links busy through fine-grained PCIe allocation, enables bidirectional transfers leveraging PCIe/NVLink bandwidth asymmetry, and balances fairness and completion time with a size-aware scheduling. We fully implement TurboBus and our experiments show that it reduces first-token latency by up to 40% for on-demand model loading (within 5% of the analytical optimum), achieves up to 1.6× throughput for KV-cache-offloaded inference, and accelerates training by up to 7%, while imposing less than 1% overhead on co-located workloads.

CCS Concepts

• **Hardware** → **Buses and high-speed links**; • **Computer systems organization** → *Heterogeneous (hybrid) systems*; *Cloud computing*; • **Computing methodologies** → Machine learning.

Keywords

PCIe bandwidth pooling, GPU memory offloading, scale-up fabric, GPU multi-tenancy, LLM systems

ACM Reference Format:
Xinyu Yang, Kaiqiang Xu, Kai Chen. 2026. TurboBus: Pooling PCIe Bandwidth for LLM Workloads via Scale-Up Fabrics. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3789240.3829130

1 Introduction

Large language models (LLMs) require substantial GPU memory, making capacity a critical bottleneck. Memory offloading addresses this by placing weights, optimizer states, or KV caches in host memory, enabling cost-efficient deployment [7, 21, 34, 35, 37, 48, 51].

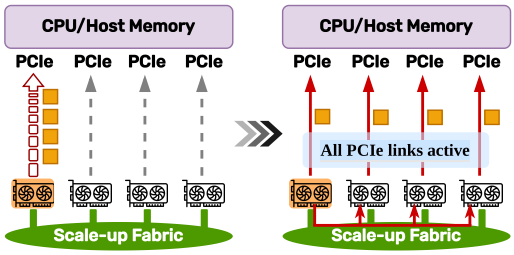


Figure 1: TurboBus pools PCIe bandwidth for GPU memory offloading. It relays data through neighboring GPUs over the scale-up fabric, allowing an offloading workload to borrow idle PCIe links instead of being limited by its own link.

However, offloading shifts the bottleneck to the CPU–GPU interconnect: FlexGen [37], which offloads KV caches for inference, spends 90% of time on PCIe transfers, while ZeRO-Offload [35], which offloads optimizer states for training, spends 24% of time on data movement. Paradoxically, these PCIe-bottlenecked applications leave over 60% of link bandwidth idle (details in §2.1).

This paradox arises at two levels: *within* a distributed job and *across* co-located jobs. Within a job, GPUs transfer in alternating phases, leaving some links idle while others are saturated. For example, ZeRO-Offload partitions optimizer states across ranks, so each GPU offloads only its assigned gradients; transfers occur at staggered times rather than simultaneously (see §2.2). Across jobs, a PCIe-bound job may saturate its link while a non-PCIe-bound neighbor's link remains underutilized. Multi-tenant GPU servers are pervasive: production clusters routinely share nodes [8, 26, 46] and cloud providers default to shared tenancy [1, 2, 39]. In both cases, some GPUs saturate their PCIe links while neighboring links sit idle. The root cause of this uneven utilization is that each GPU can only access its own PCIe link. This creates untapped opportunities for pooling PCIe bandwidth across GPUs.

Pooling is made practical by the path diversity of modern GPU servers. GPUs are connected by two kinds of links: each GPU reaches host memory via its own 16–64 GB/s PCIe link, and GPUs connect each other via a scale-up fabric such as NVLink, Infinity Fabric, or UALink at 150–450 GB/s [4, 9, 10, 14]. A GPU can therefore forward data through the scale-up fabric to neighbor GPUs and borrow their idle PCIe links (as illustrated in Fig. 1).

Prior work has explored multi-link transfers in specific scenarios, but none of them supports generic GPU memory offloading. DeepPlan [27] and Torpor [51] aggregate intra-job PCIe bandwidth for model loading but only support Host-to-Device (H2D) transfers. This excludes bidirectional workloads common in offloading; extending support is non-trivial due to H2D/Device-to-Host (D2H) contention on shared paths. memHarvester [8] enables multi-path data transfer through a tiered memory hierarchy (local GPU, neighbor GPU, host), but relies on page-fault-driven migration designed

SIGCOMM '26, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/26/08
https://doi.org/10.1145/3789240.3829130

for workloads with repeated data access. In contrast, offloading performs large, direct GPU–CPU transfers; multi-level indirection and page fault overhead add unnecessary latency. Moreover, all of them rely on store-and-forward, completing one link's transfer before starting the next, which lowers PCIe utilization and increases latency. None of them schedules concurrent link sharing across jobs, where transfers contend for the same paths (see §2.3).

In this paper, we present TurboBus (§3) to pool PCIe bandwidth across GPUs in a scale-up domain for generic GPU memory offloading (Fig. 1). To the best of our knowledge, this work is among the first to make such pooling general. Existing work [27, 51] only serves intra-job H2D model loading, whereas TurboBus works for arbitrary offloading workloads in both transfer directions and across job boundaries. However, realizing this generality raises several key challenges (§2.3). A job cannot use other job's GPUs as relays without breaking process isolation. Even with the isolation problem resolved, store-and-forward relaying buffers entire transfers on relay GPUs, inflating memory use and latency. Pipelining successive transfers could improve link utilization, but diverse transfer sizes create unbalanced stages that still render links idle. Additionally, bidirectional transfers share NVLink paths; serializing them underutilizes links, while concurrent access risks interference. Large and small transfers in the same direction also compete, risking head-of-line blocking or starvation.

To address the above challenges, TurboBus enables cross-job relay while preserving process isolation, using a privileged per-node daemon that orchestrates the transfer requests submitted by unprivileged jobs (§3.1). Rather than store-and-forward relaying, TurboBus segments each transfer into fixed-size blocks and streams them through a pipeline, keeping relay buffer bounded and latency low. Such block-level granularity further enables TurboBus to allocate paths across concurrent transfers, addressing pipeline imbalance while fully utilizing all links despite diverse transfer sizes (§3.2). For bidirectional transfers, we exploit the bandwidth asymmetry between scale-up fabric and PCIe to enable simultaneous H2D and D2H transfers on shared NVLink paths for higher aggregate throughput, and synchronize their launches to avoid interference. Along each direction, TurboBus applies size-aware scheduling: it chunks large transfers to interleave small ones for fairness, and admits large transfers exclusively to improve mean completion time (§3.3).

We have implemented TurboBus as a daemon paired with a lightweight client library (§3.4) to pool intra-node PCIe bandwidth (§4). Our evaluation spans three representative offloading workloads: on-demand model loading, KV-cache-offloaded inference, and memory-constrained training. For on-demand model loading, TurboBus reduces time-to-first-token (TTFT) by up to 40%, within 5% of the analytical optimum. KV-cache-offloaded inference achieves up to 1.6× throughput, while memory-constrained training sees up to 7% faster iterations. Meanwhile, a co-located workload whose idle link is borrowed sees less than 1% overhead (§5).

In summary, this paper makes the following contributions:

- We identify a PCIe utilization paradox (§2): offloading workloads are PCIe-bottlenecked yet leave over 60% of link capacity idle, within jobs due to alternating transfer phases and across jobs due to uneven transfer demand.
- We present TurboBus (§3), which generalizes PCIe bandwidth pooling across co-located jobs via a privileged daemon that transparently routes data through idle links while preserving process isolation. TurboBus combines pipelined relay execution, which streams blocks through relay GPUs with bounded memory to fill idle links, with cross-job concurrency scheduling, which runs both directions concurrently within NVLink's headroom and fairly schedules transfers.
- We implement a prototype (§4) of TurboBus and the evaluation (§5) shows that TurboBus achieves up to 40% TTFT reduction (within 5% of the analytical optimum), up to 1.6× serving throughput, and up to 7% faster iterations, while imposing less than 1% overhead on co-located jobs.

2 Background and Motivation

Modern multi-GPU servers (*nodes*) connect each GPU to CPU memory (or *host memory*) via PCIe and to peer GPUs via a scale-up fabric. On such nodes, offloading workloads are PCIe-bottlenecked yet leave over 60% of link bandwidth idle, both within and across jobs, creating opportunities for pooling PCIe bandwidth through the fabric. This section characterizes offloading workloads and link underutilization (§2.1), analyzes opportunities for bandwidth pooling and the enabling scale-up fabric architecture (§2.2), and discusses gaps in existing work and the challenges they leave open (§2.3).

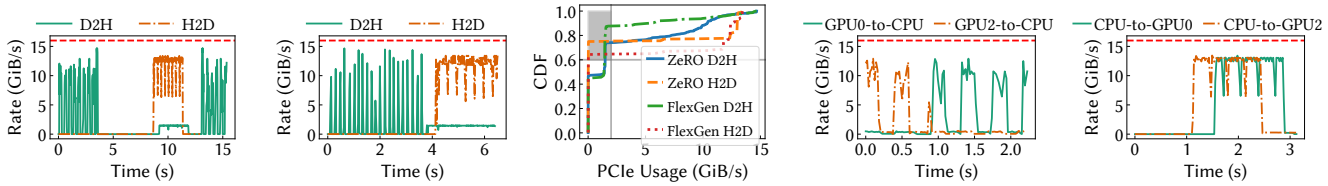
2.1 PCIe-Bottlenecked Workloads

LLM systems widely leverage heterogeneous CPU–GPU resources to balance performance and cost. CPU memory is commonly used to extend the fast but scarce GPU memory, a practice known as *GPU memory offloading*. We identify three representative categories of such applications:

- **On-demand model loading:** Models are cached in host memory and loaded into the GPU on-demand to serve requests, reducing GPU memory pressure for multi-model serving [5, 21, 27, 45, 48, 51].
- **KV-cache-offloaded serving:** The key-value (KV) cache for attention is offloaded to host memory and loaded on-demand during token generation, allowing larger batch sizes and longer sequences [29, 30, 37, 53].
- **Memory-constrained training:** Optimizer states and gradients are offloaded to host memory, enabling training of larger models on limited GPU memory [20, 34, 35].

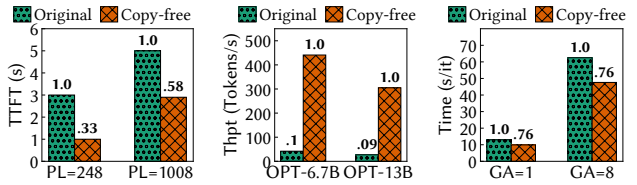
These workloads all trade PCIe transfer overhead for reduced GPU memory consumption and often become PCIe-bottlenecked. Yet, paradoxically, these PCIe links sit idle much of the time. We next quantify both phenomena.

2.1.1 Performance Impact of Data Transfer. To isolate the impact of CPU–GPU data transfer, we perform a *what-if* analysis: we compare the same representative workloads in two configurations, *original* (normal execution) and *copy-free* (memory copy operations replaced with no-ops). The performance gap between the two quantifies the overhead attributable to data transfer. These workloads move large, contiguous buffers between pinned host memory and GPU memory. Because the host memory is pinned, each transfer is a direct DMA with no intermediate host-side copy, and host memory bandwidth far exceeds PCIe bandwidth. The remaining per-transfer



(a) ZeRO-Offload Traffic (b) FlexGen Traffic (c) Utilization CDF (d) D2H (DP=2) (e) H2D (DP=2)

Figure 2: (a–b) PCIe traffic of ZeRO-Offload and FlexGen shows bursty, phase-driven usage with idle periods; (c) CDF of PCIe utilization: bandwidth stays at or below 2 GB/s for over 60% of time; (d–e) ZeRO-Offload with DP=2: transfers across ranks are offset in time.



(a) PipeSwitch (b) FlexGen (c) ZeRO-Offload

Figure 3: What-if analysis comparing *original* vs. *copy-free* execution, showing data transfer overhead on (a) TTFT, (b) throughput, and (c) iteration time.

setup costs (e.g., kernel launch and DMA setup) are fixed and negligible for such large transfers. This overhead is therefore dominated by the PCIe transfer.

On-demand model loading. We evaluate PipeSwitch [5], which pipelines model loading with inference at layer granularity: as each layer's weights are loaded from CPU to GPU, inference proceeds on already-loaded layers. Using Llama-2-13B on an NVIDIA V100 GPU, we measure TTFT for prompt lengths of 248 and 1008 tokens. As shown in Fig. 3 (a), data transfer increases TTFT to 1.7–3.0× that of copy-free, as model loading dominates the critical path.

KV-cache-offloaded serving. We evaluate FlexGen [37], which offloads the KV cache to CPU memory. In transformer inference, attention requires key-value vectors for all preceding tokens; caching these vectors avoids redundant computation but consumes substantial memory (the KV cache can even exceed model weight size [37]). FlexGen addresses this by offloading the KV cache: during the *prefill* stage, KV vectors are computed and moved to CPU; during *decoding*, they are loaded back on-demand for each token generation. Using OPT models on a V100 GPU with batch size 32 and prompt length 512, we measure decoding throughput. Fig. 3 (b) shows that KV cache offloading reduces throughput to approximately 1/10th of the copy-free baseline, though it remains faster than recomputing KV vectors at each step.

Memory-constrained training. ZeRO-Offload [35] offloads optimizer states and gradients to CPU memory. During training, gradients computed on the GPU are transferred to the CPU (D2H), where the optimizer updates weights, which are then copied back to the GPU (H2D). To enable larger effective batch sizes, ZeRO-Offload is often combined with *gradient accumulation* (GA): multiple forward-backward passes accumulate gradients before a weight update. With GA, each non-update step transfers accumulated gradients to the GPU, computes new gradients, and transfers the sum back, introducing repeated data transfers. Fine-tuning Llama-3-8B on a V100 GPU with GA steps of 1 and 8, we compare standard training against a copy-free version. Fig. 3 (c) shows that data transfer degrades iteration time by up to 24%.

Table 1: Average PCIe utilization remains below 27% of the 16 GB/s link capacity.

Application	Direction	Util.
ZeRO-Offload	H2D	19.22%
	D2H	17.82%
FlexGen	H2D	26.78%
	D2H	8.39%

2.1.2 Characterizing PCIe Underutilization. We measure PCIe traffic using NVIDIA Management Library (NVML) [18] APIs to characterize link utilization.

Traffic patterns reveal periodic usage. Fig. 2 (a) shows the PCIe traffic for ZeRO-Offload fine-tuning Llama-3-8B [33]. The pattern consists of alternating phases: D2H during gradient transfer, an idle period during CPU-side optimizer computation, and H2D when updated weights return to the GPU. This cyclical pattern leads to periodic, rather than continuous, link usage. Similarly, Fig. 2 (b) shows FlexGen traffic divided into prefill and decode stages. The prefill stage shows a burst of D2H traffic as KV vectors move to CPU memory. The decode stage is dominated by H2D traffic as KV cache is loaded for each token, with small concurrent D2H for newly generated KV vectors.

Utilization is far below capacity. Table 1 summarizes average PCIe utilization: ZeRO-Offload achieves only 17%–19% and FlexGen only 8%–27% of the 16 GB/s PCIe 3.0 x16 capacity. The CDF in Fig. 2 (c) reveals why: for more than 60% of the time, bandwidth usage does not exceed 2 GB/s (12.5% utilization). The bursty traffic patterns leave links idle between transfer phases. This substantial idle capacity is bandwidth that could be reclaimed. §2.2 shows why it is poolable: transfers are offset across ranks within a job, and demand is uneven across co-located jobs.

2.2 Opportunities for Bandwidth Pooling

The idle periods identified above create pooling opportunities at two levels, within a single job and across co-located jobs.

2.2.1 Intra-Job Opportunities. Within a single distributed job, transfers interleave across ranks. In data-parallel training with ZeRO-Offload, each rank owns the optimizer states for a different subset of the model's layers. Because the backward pass produces gradients layer by layer, the ranks' D2H offloads for their assigned gradients are staggered, not simultaneous (see Fig. 2 (d)). A similar pattern

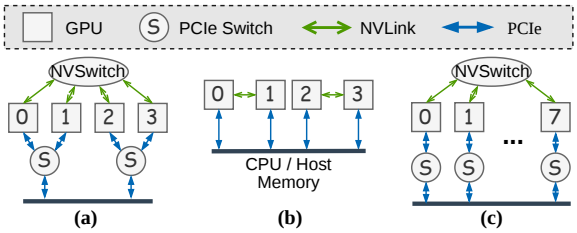


Figure 4: Scale-up fabrics are prevalent in modern multi-GPU servers: (a) DGX V100 and (c) H800 use a switched fabric, while (b) RTX 3090 uses a direct mesh.

occurs for H2D traffic during loading updated weight to GPUs, as CPU-bound optimizer computation creates a processing queue that offsets H2D operations (see Fig. 2 (e)).

2.2.2 Cross-Job Opportunities. Modern GPU servers have a fixed scale-up topology (e.g., 8 GPUs on DGX [14], 72 GPUs on NVL72 [16]), but user demands are diverse. Many workloads require only a fraction of a node: single-GPU applications such as FlexGen, small data-parallel jobs using 2–4 GPUs, or fine-tuning tasks with limited memory needs. This mismatch drives multi-tenancy from both sides: providers partition nodes to maximize sold GPU-hours [39], while users rent partial allocations to minimize cost [1, 2]. Production clusters exhibit the same pattern, with schedulers co-locating jobs to improve utilization [26, 46].

Co-location creates opportunities for bandwidth pooling in two ways. When a PCIe-bound offloading job shares a node with non-PCIe-bound workloads, the co-located GPUs' PCIe links stay largely idle, free for it to borrow. When multiple offloading jobs share a node, each uses PCIe only in bursts, so even co-located PCIe-heavy jobs leave idle capacity to pool (§5.3.2 confirms this).

2.2.3 Enabling Architecture. To exploit these opportunities, GPUs must share access to each other's PCIe links. The scale-up fabric enables this by giving each GPU an alternative path to host memory through its neighbors. Its high bandwidth keeps the relay hop from becoming the bottleneck. On our V100 testbed, the NVLink between a GPU pair provides 50 GB/s per direction, far exceeding PCIe's 16 GB/s [10]. This gap is not specific to our V100 testbed; it persists and widens on newer GPUs (e.g., H100 [14], B200 [16]), so the relay hop stays hidden and relaying remains effective on modern hardware (§7). Similar fabrics include AMD's Infinity Fabric and the upcoming UALink [4, 9]. The fabric thus serves as a relay backplane. As shown in Fig. 4 (a), GPU0 can transfer data to host memory via GPU2's PCIe link by first forwarding through NVLink, effectively borrowing GPU2's idle bandwidth.

2.3 Gaps and Challenges

Table 2 compares existing PCIe bandwidth pooling approaches. DeepPlan [27] and Torpor [51] both aggregate PCIe bandwidth via NVLink relay and pipeline model loading with inference. DeepPlan profiles per-layer execution to select between direct host access and preloading for each layer. Torpor pools GPU memory across a multi-GPU node and schedules transfer paths for heavy versus light models to meet latency SLOs. memHarvester [8] targets workloads with temporal and spatial data locality, evicting memory pages to neighbor GPU memory (via NVLink) before host memory (via PCIe) to reduce re-access latency. It is not designed for the direct and large CPU–GPU transfers common in offloading. All three use store-and-forward that serializes PCIe and NVLink phases, and none schedules concurrent bidirectional transfers across jobs.

Table 2: Comparison of PCIe bandwidth pooling approaches. TurboBus provides two offloading-specific features: pipelined relay (bounded memory and low latency with fine-grained allocation) and concurrency scheduling (concurrent bidirectional transfers for higher throughput with fairness and bounded blocking).

	mem-Harvester	DeepPlan /Torpor	**TurboBus**
Scope			
Intra-job	✓	✓	✓
Inter-job	✓	✗	✓
Direction			
Host-to-Device	✓	✓	✓
Device-to-Host	✓	✗	✓
Offloading-Specific			
Pipelined Relay	✗	✗	✓
Concur. Sched.	✗	✗	✓

Realizing general, efficient, offload-specific PCIe pooling across jobs is non-trivial. We identify five key challenges, each addressed by a corresponding design component in §3:

C1: Cross-job isolation. In multi-tenant environments, GPUs belong to different jobs with separate address spaces and permissions. Process isolation prevents a job from accessing another job's GPU (including its memory and PCIe link) to use as a relay.

C2: Relay overhead. Naive store-and-forward relaying buffers the entire payload on the relay GPU before forwarding, serializing PCIe and NVLink phases. This increases transfer latency and, for large payloads (e.g., multi-GB model weights), exhausts GPU memory.

C3: PCIe link underutilization. A natural approach to optimize link utilization is inter-transfer pipelining, overlapping one transfer's relay phase with another's PCIe phase so the link stays continuously busy. However, offloading workloads exhibit diverse transfer sizes, where multi-GB model weights, fine-grained KV cache blocks, and per-layer gradients vary by orders of magnitude. This diversity introduces pipeline bubbles: shorter transfers finish early, leaving paths idle until longer ones complete (as shown in Fig. 6 (b), left).

C4: Concurrent bidirectional transfers. General offloading runs concurrent H2D and D2H transfers that share NVLink paths. For example, GPU0 offloading (D2H) through GPU2 and GPU2 loading (H2D) through GPU0 both use the NVLink from GPU0 to GPU2 (Fig. 6 (a)). Serializing them wastes PCIe bandwidth, while sharing naively risks interference, so the challenge is to let both directions share NVLink safely.

C5: Blocking and starvation. Concurrent transfers from different jobs compete for shared paths. Pure FCFS causes head-of-line (HoL) blocking: large transfers delay small latency-sensitive ones. Prioritizing small transfers avoids this but can starve large transfers indefinitely. A practical policy must bound blocking for small transfers while ensuring large transfers complete fairly.

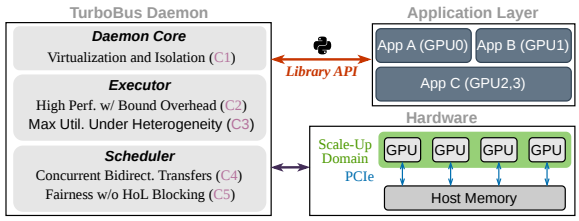


Figure 5: TurboBus architecture. Clients issue transfer requests through a transfer API to a privileged daemon comprising three components: daemon core for cross-job isolation (C1), executor for pipelined relay execution (C2, C3), and scheduler for concurrency scheduling (C4, C5).

3 Design of TurboBus

3.1 Overview

TurboBus addresses the challenges identified in §2.3 through a client-server architecture, as illustrated in Fig. 5.

Cross-job isolation (**C1**) is an architectural challenge: process boundaries prevent jobs from accessing each other's GPUs. We address this through a privileged per-node daemon that has access to all GPUs and exposes an API for offloading jobs to submit transfers. Offloading jobs act as clients: they first establish a connection with the daemon, then allocate host and device tensors for data movement. These tensors are shared with the daemon via zero-copy mechanisms, enabling direct data operations. When a client issues a transfer request, the daemon enqueues it for scheduling. The *scheduler* determines the order in which requests are served. The *executor* then splits each request across a set of paths between the GPU and host memory and streams the data through them: a direct path over the GPU's own PCIe link, and relay paths that hop over NVLink to a neighbor and through its PCIe link. The direct link is thus one of the pooled paths, not a separate fast path. Because a client only issues requests through the API, it never accesses another job's GPU; the privileged daemon performs the relays on its behalf. This achieves cross-job link access while preserving process isolation (implementation details in §4).

The following subsections detail the executor, scheduler, and user-facing library. The executor (§3.2) addresses relay overhead and path underutilization (**C2, C3**) with pipelined block streaming and block-level path allocation. The scheduler (§3.3) runs both directions concurrently and orders competing transfers (**C4, C5**). Finally, a Python library (§3.4) wraps the transfer API, requiring only minimal code changes to integrate with existing applications.

3.2 Pipelined Relaying

Store-and-forward relaying buffers entire transfers, inflating memory and latency (C2). Pipelining transfers would keep links busier, but their diverse sizes create bubbles that leave paths idle (C3). Both stem from processing transfers at a coarse, whole-transfer granularity. We instead decompose transfers into fixed-size blocks. Block streaming pipelines these blocks through relays with bounded memory, while block-level allocation packs blocks from concurrent transfers to fill all paths.

3.2.1 Block Streaming. We segment transfers into fixed-size *blocks* and stream them through the relay GPU. Once the relay GPU receives a block via NVLink, it immediately forwards that block via PCIe while simultaneously receiving the next (Fig. 6 (b), middle); the H2D path mirrors this, with NVLink and PCIe reversed. Since NVLink is faster than PCIe, the relay GPU finishes receiving the next block before it finishes transmitting the current one, fully hiding the NVLink latency. This pipelined design yields three benefits:

- **Bounded memory**: The relay buffer is fixed at a small multiple of the block size, independent of transfer size.
- **Reduced latency**: Transfer completion time approaches PCIe-only time, effectively hiding the NVLink hop.
- **Full utilization**: The PCIe link stays busy with no idle periods waiting for NVLink. Uniform block duration also enables bubble-free pipelining across transfers.

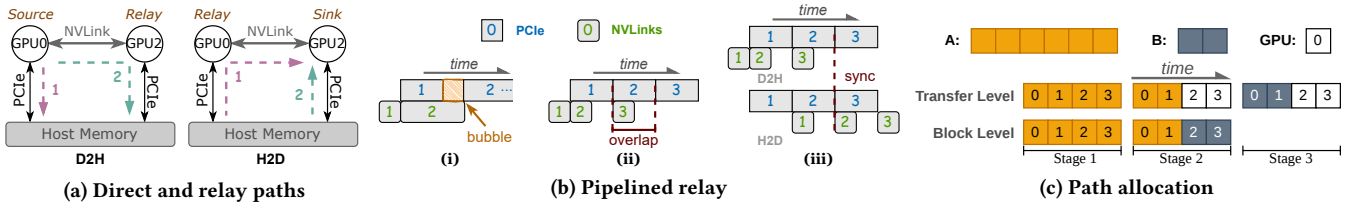
Block size selection. The throughput-optimal block size depends on the hardware, not the transfer, so a single block size suits all transfers large enough to pipeline; very small ones are bypassed. The block size involves a trade-off between PCIe efficiency and overhead: larger blocks raise efficiency by amortizing the fixed per-block launch cost, but enlarge the relay buffers, lengthen warm-up latency, and increase internal fragmentation. We profile this trade-off and pick the smallest block size that reaches the throughput peak, minimizing buffer memory and warm-up latency without sacrificing PCIe efficiency. On our V100 testbed the block size is 16MB; §5.2 shows the throughput curve and validates the choice across PCIe generations.

Bypassing small transfers. Routing small transfers through the relay pipeline underutilizes path bandwidth and incurs scheduling overhead. An alternative is bypassing the relay to use native `cudaMemcpy` directly, but this creates unscheduled background traffic that can interfere with pipelined transfers. To balance these concerns, we bypass transfers below half the block size (8MB on our V100 testbed): for these, pipelining overhead exceeds its benefit, and their small volume barely perturbs scheduled transfers.

Buffer sizing. The relay pipeline overlaps receiving and forwarding, so the relay GPU keeps two buffers: one fills while the other drains. Supporting both H2D and D2H directions doubles this to four buffers, yielding a total relay overhead of $4 \times 16\text{MB} = 64\text{MB}$, a negligible fraction of GPU memory that remains constant regardless of transfer size.

3.2.2 Block-Level Allocation. Block streaming splits each transfer into an integer number of blocks. If we allocate at transfer granularity, a transfer's final blocks rarely fill all paths, so some paths sit idle. With the large number of transfers in LLM offloading, this remainder waste accumulates significantly. Uniform block sizes enable a simple solution: we allocate at block granularity, packing blocks from different transfers to fill the gaps.

Consider the transfer-level case in Fig. 6 (c): four paths process Transfer A (6 blocks) followed by Transfer B (2 blocks). With transfer-level allocation, A fills all four paths in stage 1, but its last two blocks use only two paths in stage 2, leaving the other two idle. B must wait until A completes even though those two paths sit idle, so its ready blocks cannot fill them.



(a) Direct and relay paths (b) Pipelined relay (c) Path allocation

Figure 6: Executor mechanisms. (a) Direct and relay paths for D2H and H2D. Path 1 is direct for D2H but relay for H2D; path 2 is the reverse. (b) Pipelined relay: bubbles from varying sizes (left); block streaming (middle); synchronized bidirectional (right). And (c) Path allocation: transfer-level leaves paths idle; block-level fully utilizes all paths.

With block-level allocation (Fig. 6 (c), block-level), the allocator examines pending blocks from *all* active transfers and assigns each available path to a ready block at each scheduling round. The two paths idle in A's second stage are immediately assigned to B's blocks. By packing blocks from concurrent transfers, all paths remain fully utilized regardless of individual transfer sizes.

3.2.3 Summary. Both mechanisms share a common insight: operating at block granularity rather than transfer granularity. Block streaming pipelines fixed-size blocks through relays, bounding relay memory and hiding the NVLink hop (C2). Fixed-size blocking, however, does not solve link underutilization (C3) but only changes its form: the diverse transfer sizes that caused pipeline bubbles now leave some paths idle in the last stage. Block-level allocation resolves it by allocating paths across concurrent transfers, fully utilizing all links. Together, they achieve near-optimal bandwidth utilization with bounded 64MB memory overhead. We evaluate both mechanisms in §5.2.

3.3 Concurrency Scheduling

The scheduler manages shared link capacity at two levels: between the two directions, it runs them concurrently (C4); within each direction, it bounds blocking and starvation (C5) while also improving completion time. For bidirectional traffic, it exploits NVLink's bandwidth headroom over PCIe to run H2D and D2H concurrently on the same link at full PCIe rate. Within each direction, it exploits the bimodal distribution of transfer sizes, chunking large transfers to bound blocking and token-gating concurrent large transfers to avoid large-large interference.

3.3.1 Scheduling for Bidirectional Traffic. Concurrent H2D and D2H transfers can share the same NVLink path (Fig. 6 (a)). Intuition suggests the two directions should not share an NVLink, yet we find they can share it at no cost: at most two PCIe-rate transfers ever use it at once, and NVLink exceeds twice the PCIe rate, so both run concurrently without slowing each other.

Strawman: serializing the two directions. The obvious approach is to give the NVLink to one direction at a time. This approach, however, complicates scheduling and, more importantly, leads to a significant waste of PCIe bandwidth. To keep them separate in Fig. 6 (a), the scheduler would have to delay one direction, forcing an available PCIe link to remain idle.

Concurrent bidirectional transfers. Rather than serializing them, we let H2D and D2H share the same NVLink, and show it always has room for both. By design, relays use only a single NVLink hop, and each PCIe direction carries at most one request at a time. Multi-hop forwarding is avoided because it cannot reach PCIe links beyond direct neighbors and only adds NVLink latency without exposing new PCIe bandwidth. Under these two constraints, at most two simultaneous transfers can share the NVLink bandwidth from GPU i to j (e.g., GPU0 to GPU2 in Fig. 6 (a)): one is j's H2D request relayed by i, the other is i's D2H request relayed by j. Both traverse the NVLink from i to j at the same time. Since each path is bottlenecked by the slower PCIe link, the NVLink bandwidth requirement is at most twice the PCIe bandwidth.

A key architectural characteristic provides the headroom: NVLink bandwidth is typically more than double that of PCIe. On our V100 testbed (Fig. 4 (a)), unidirectional NVLink provides 50 GB/s versus PCIe's 16 GB/s. This property also holds for A100 and H100 systems [10, 14, 15]. Consequently, even if two transfers share an NVLink and halve its effective bandwidth, the halved bandwidth still exceeds PCIe. As a result, neither direction has to give up the shared NVLink, so both stay active, raising aggregate throughput despite the per-transfer overhead (as shown in Fig. 8 (b)).

Practical issue: synchronization interference. The analysis above motivates running H2D and D2H transfers concurrently; since they use separate copy engines [13], it is natural to drive each direction from its own thread, each synchronizing independently. However, we observed severe performance interference on our testbed, with throughput dropping by up to 40% compared to single-direction transfers. We hypothesize that this arises from CUDA runtime's internal locking: a synchronization call on one stream (e.g., `cudaStreamSynchronize`) inadvertently blocks other streams, even when they use independent copy engines.

Synchronized pipelining. To mitigate this interference, we employ a single control thread to launch all transfer operations. Instead of synchronizing each stream independently, we divide execution into fixed-duration time slots (each matching the block transfer time) and synchronize all active streams together at slot boundaries (Fig. 6 (b), right). By batching synchronization into the single thread, we avoid the locking, so both directions run at full PCIe rate, raising aggregate bandwidth.

Coexisting with other NVLink traffic. By design, TurboBus schedules only CPU–GPU transfers, not device-to-device (D2D) traffic. The application's own D2D, such as NCCL collectives, thus runs over the NVLink outside TurboBus's control. A relay needs only PCIe-rate traffic on the NVLink, far below its bandwidth, so unless the D2D saturates the link, the relay still gets its share and keeps its PCIe rate. Conversely, the relay's NVLink traffic may slow the D2D, but when the job is PCIe-bound, its slowdown stays hidden behind its PCIe bottleneck. Our ZeRO-Offload runs sustain a 4%–7%

gain while running data-parallel all-reduce over NVLink alongside relay (§5.3). Across jobs, co-located jobs do not communicate, so a cross-job relay path carries no competing D2D traffic. If the D2D instead saturates NVLink or lies on the critical path, the relay can slow it and the benefit shrinks, a case we leave to future work.

3.3.2 Scheduling Policy. Within each direction, concurrent transfers compete for shared paths. GPU memory offloading exhibits *bimodal* transfer sizes: large transfers (multi-GB model weights, optimizer states) and small transfers (fine-grained KV cache blocks, per-layer gradients). We classify transfers at admission using a size threshold and apply tailored scheduling.

Strawman: FCFS and shortest-first. FCFS at transfer granularity causes head-of-line blocking: a multi-GB transfer monopolizes all paths for a long time, delaying small transfers behind it, the classic convoy effect [6]. Prioritizing small transfers avoids the convoy effect but can starve large transfers indefinitely, as newly arriving small transfers continually preempt them.

Chunking large transfers. To bound blocking without starving large transfers, we partition large transfers into fixed-size chunks and schedule chunks alongside small transfers in FCFS order. Crucially, each large transfer maintains *at most one outstanding chunk*: it enqueues the next chunk only after the current one completes. This converts unbounded blocking (an entire large transfer) into bounded blocking (at most one chunk time), ensuring small transfers wait no longer than a single chunk duration.

Token-gated admission. Chunking bounds blocking for small transfers and avoids starving large ones, but does not prevent *large-large interference*: serving multiple large transfers concurrently delays all of them, increasing mean completion time. We introduce a token mechanism that serializes large transfers, permitting only one to enqueue chunks at a time. The token is granted in arrival order, preserving FCFS among large transfers. Small transfers bypass the token and proceed directly.

3.3.3 Summary. Running H2D and D2H concurrently exploits NVLink's headroom over PCIe, raising aggregate throughput. Within each direction, chunking provides fairness between large and small transfers, while token-gating serializes large transfers to avoid large-large interference. We evaluate these mechanisms in §5.2, showing that running the two directions concurrently yields higher aggregate throughput than serializing them, and that chunking and token-gating both cut completion time.

3.4 TurboBus library

The TurboBus library exposes a high-level Python API (Table 3) designed for seamless integration into existing applications. To leverage TurboBus, users install it as a standard package via `pip` and substitute their native tensor allocation and copy operations with TurboBus equivalents (Fig. 7).

The API is organized into three groups. *Session management*: `init()` establishes connections with the TurboBus daemon; `clean()` disconnects and releases resources. *Tensor allocation and destruction*: `create_host_tensor()` and `create_dev_tensor()` allocate host or device tensors for a specified GPU, returning PyTorch-compatible tensor objects; their `destroy_*` counterparts deallocate these tensors. *Data movement*: `pair()` binds a host tensor and a device tensor into a `MemPair` object, which exposes `to_host()` and `to_dev()` methods to trigger D2H and H2D transfers, respectively. At runtime, these API calls are transparently delegated to the daemon, which orchestrates multi-path data transfers.

Table 3: TurboBus API: session management, tensor allocation/deallocation, and data movement.

`init()`, `clean()`

`create_host_tensor(dev_id, shape, dtype)->Tensor`
`create_dev_tensor(dev_id, shape, dtype)->Tensor`
`destroy_host_tensor(tensor)`
`destroy_dev_tensor(tensor)`

`pair(host_tensor, dev_tensor)->MemPair` `MemPair.to_host()`, `MemPair.to_dev()`

```python
import torch

# Create tensors
cpu_t=torch.ones(256,1024,1024)
gpu_t=torch.zeros(256,1024,1024,
              device='cuda:0')

# Non-blocking data transfer
gpu_t.copy_(cpu_t,
            non_blocking=True)
# Sync
torch.cuda.synchronize()
print("Copy finished.")
```

(a) Original code

```python
import torch
import turbobus as tb
tb.init() # initialize
# Create tensors
cpu_t=tb.create_host_tensor(
            0, (256,1024,1024))
gpu_t=tb.create_dev_tensor(
            0, (256,1024,1024))
# Create memory pair and copy
mp=tb.pair(cpu_t,gpu_t).to_dev()
torch.cuda.synchronize()
print("Copy finished.")
tb.clean() # clean up
```

(b) Code using TurboBus

Figure 7: Integrating TurboBus requires minimal changes: adding initialization/cleanup calls and replacing tensor allocation and copy operations with TurboBus equivalents.

Ease of integration. As illustrated in Fig. 7, integration is localized: the application's control flow, computation kernels, and overall structure remain unchanged, and only session management, tensor allocation, and data movement use TurboBus's API. This design enables incremental adoption, allowing developers to accelerate specific data paths (e.g., model loading or KV cache swapping) without refactoring their entire codebase.

4 Implementation

TurboBus comprises approximately 3,900 lines of code: a C++ daemon (~2,500 LoC), a C++ client library (~1,100 LoC), and a Python frontend (~300 LoC). We implement the client-server architecture described in §3.1 to address isolation (C1). A privileged daemon process acts as the server with visibility over all GPUs, while unprivileged applications interact through a client library distributed as a Python package.

Zero-copy memory sharing. A central challenge in this multi-process architecture is enabling zero-copy memory sharing between the client application and the daemon. We employ distinct mechanisms for host and device memory. For host memory, the client library allocates from the system's shared memory filesystem (`/dev/shm` on Linux); the daemon memory-maps this region for direct access. For GPU memory, we leverage CUDA's IPC primitives: the client obtains an opaque handle via `cudaIpcGetMemHandle()`,

which the daemon opens to access the same device memory region. These two mechanisms leave the UNIX domain socket for lightweight control-plane communication only.

Topology abstraction. To support diverse server configurations, TurboBus abstracts PCIe topology into generalized data structures: *link groups* (sets of PCIe links that can be pooled), *GPU-to-link mappings* (direct PCIe path for each GPU), and *relay capability lists* (which GPUs can serve as relays for each link). At startup, the daemon takes the system topology (e.g., from `nvidia-smi topo -m`) and parses connection types. NVLink-connected GPUs form a link group suitable for relay. GPUs connected only through a shared PCIe switch are placed in a separate group, since relaying through them shares the same upstream PCIe link and exposes no additional bandwidth. This lets TurboBus support different hardware configurations without code changes.

Daemon architecture. The daemon employs a two-thread design to maximize throughput. A *listener thread* uses edge-triggered epoll on the UNIX socket for low-latency request reception, with CPU affinity pinning to reduce jitter. A *worker thread* executes the scheduling and transfer logic, housing two modules:

- The *Scheduler* module implements the concurrent bidirectional transfer and request prioritization policies (§3.3), with configurable baselines (FCFS, shortest-job-first, fair-share) for evaluation.
- The *Executor* realizes the pipelined block streaming and block-level path allocation mechanisms (§3.2).

The daemon's CPU overhead is minimal, as the critical path involves only lightweight scheduling decisions and CUDA API calls rather than data copying. The scheduler re-evaluates path allocation every round, so it reacts to changes in link contention at the granularity of a single block transfer rather than through a separate control loop. Because the allocation is a simple bitmap and relays are single-hop (§3.3.1), no routing search is needed even as the domain grows. We scope TurboBus to intra-node pooling and leave relay-target selection across the many candidate neighbors in larger switched scale-up domains (e.g., NVL72) to future work. For fault tolerance, each transfer is independent and the API mirrors `cudaMemcpy` semantics, so if the daemon is unavailable, a request falls back to a direct `cudaMemcpy` on the GPU's own link, preserving correctness while forgoing the pooling speedup.

CUDA integration. To integrate with the CUDA programming model, copy requests are enqueued as asynchronous operations via `cudaLaunchHostFunc`, making them behave like native CUDA stream operations that respect existing dependency chains.

5 Evaluation

5.1 Experiment Setting

Testbeds. We use the NVIDIA DGX V100 as our primary testbed. It has four V100 GPUs with 32GB VRAM connected by NVSwitch. These four GPUs are connected to two independent PCIe switches, exposing two poolable PCIe 3.0 x16 links per direction (16 GB/s each, 32 GB/s aggregate). The logical topology is shown in Fig. 4 (a). NVLink bandwidth exceeds PCIe by more than 2× on the V100, and this asymmetry holds across modern GPU servers. To confirm our conclusions generalize beyond the V100, our microbenchmarks also run on an RTX 3090 server (PCIe 4.0, Fig. 4 (b)) and an eight-GPU H800 server (PCIe 5.0, Fig. 4 (c)), characterizing relay overhead and scaling across PCIe generations (§5.2); we discuss broader applicability in §7. All experiments run with NVIDIA MPS (Multi-Process Service) [17] enabled, which allows efficient multi-process GPU sharing, so the daemon and application processes share the GPU without added overhead. MPS is widely supported on all NVIDIA GPU architectures from Volta onward.

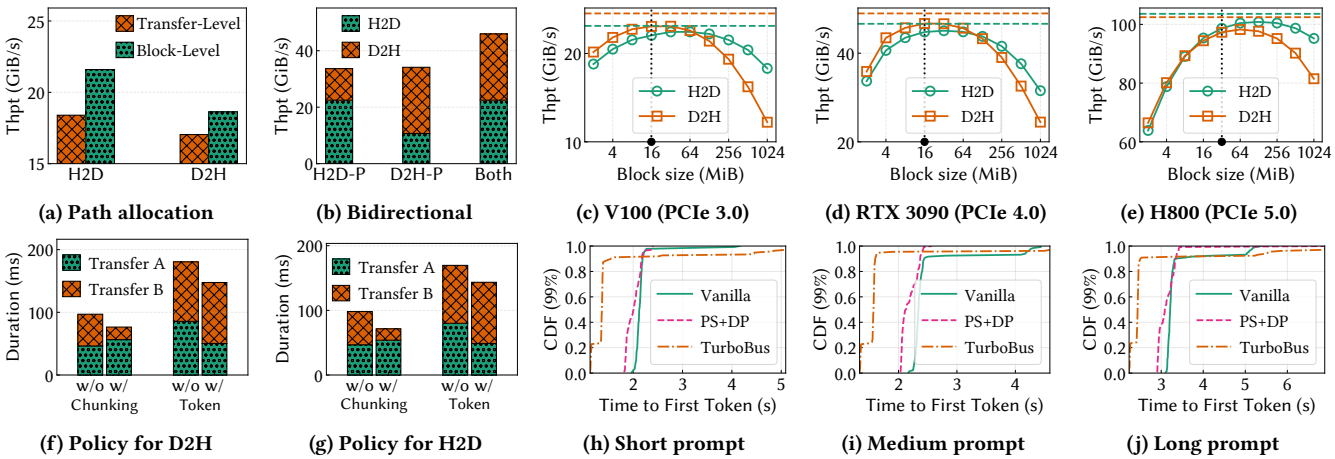
Analytical model. To quantify the potential benefit of bandwidth pooling, we use a simple Amdahl-style model. Let an application's execution time be decomposed as $T = T_c + T_d$, where T_c is computation time (GPU kernels, CPU processing, etc.) and T_d is non-overlappable data transfer time, the portion that cannot be hidden by computation (PCIe-bound). With N PCIe links available via pooling, the optimal transfer time becomes T_d/N, yielding $T_{\text{opt}} = T_c + T_d/N$. The speedup is thus bounded by:

$$\text{Speedup} = \frac{T}{T_{\text{opt}}} = \frac{1}{1 - f + f/N} \tag{1}$$

where $f = T_d/T$ is the fraction of time spent on non-overlappable data transfer. Applications with higher transfer fractions benefit more: FlexGen ($f \approx 90\%$) can achieve up to 3.0× speedup with 4 links, while ZeRO-Offload ($f \approx 24\%$) reaches only 1.2×. This model considers only non-overlappable transfers, which can be measured via the what-if analysis in §2.1.1; any overlappable portion that could also benefit from pooling is ignored. Since our targeted applications are bottlenecked by data transfer, this overlappable portion is small, making the approximation reasonable.

Evaluation methodology. Micro-benchmarks (§5.2) validate individual design components: (1) pipelined transfer with profiled buffer sizing, (2) block-level allocation, (3) concurrent bidirectional transfers, (4) request prioritization scheduling, and (5) relay scaling and NUMA sensitivity. End-to-end experiments (§5.3) evaluate three workload types: (1) on-demand model loading under both serverless emulation and isolated settings, (2) KV-cache-offloaded serving with mutual relaying and heterogeneous co-location, and (3) memory-constrained training with ZeRO-Offload.

Offloading applications. We evaluate three applications representing the offloading workloads introduced in §2.1. For on-demand model loading, we use the Llama models with layer-wise loading following PipeSwitch's approach [5], with data transfers routed through TurboBus. For KV-cache-offloaded serving, we use FlexGen, a widely used and well-maintained system that offloads the KV cache to host memory during decoding. Its H2D/D2H transfer pattern is representative of modern KV-reuse systems. For memory-constrained training, we use DeepSpeed, which supports ZeRO-Offload for offloading optimizer states to host memory. All three run with a tensor-parallelism degree of one (TP=1), matching our baselines: single-GPU FlexGen and on-demand loading, and data-parallel ZeRO-Offload. Under TP>1, TurboBus helps little within a synchronized TP group but still pools idle links across TP groups, DP replicas, and co-located jobs. For on-demand loading and FlexGen, we use the architectures each reference system supports, varying model size; for ZeRO-Offload we deliberately vary both architecture and size to show the benefit generalizes across model families. Since TurboBus's benefit is governed by the volume and



(a) Path allocation (b) Bidirectional (c) V100 (PCIe 3.0) (d) RTX 3090 (PCIe 4.0) (e) H800 (PCIe 5.0)

(f) Policy for D2H (g) Policy for H2D (h) Short prompt (i) Medium prompt (j) Long prompt

Figure 8: Micro-benchmarks and serverless TTFT: (a) block-level allocation improves throughput by 20%; (b) concurrent bidirectional transfers outperform serialization; (c–e) relay throughput vs. block size across PCIe generations, where dashed lines mark 2× the measured single-link PCIe bandwidth (the two-link ideal) and the dotted line marks the profiler-selected block size; (f–g) chunking and token-based control reduce completion time by 15%–27% (Transfer A: 1 GB; Transfer B: 32 MB for chunking, 1 GB for token); (h–j) serverless TTFT CDFs: TurboBus achieves 17%–25% median reduction across prompt lengths; the y-axis is truncated at the 99th percentile, where TurboBus trades up to 10% higher P99 for these median gains.

pattern of CPU–GPU transfers rather than the model architecture, these choices do not bias the comparison. Although we measure only these applications, TurboBus applies more broadly to workloads with substantial runtime CPU–GPU data movement, including checkpoint save/restore during fault-tolerant training [44] and hybrid inference for large models [7, 23].

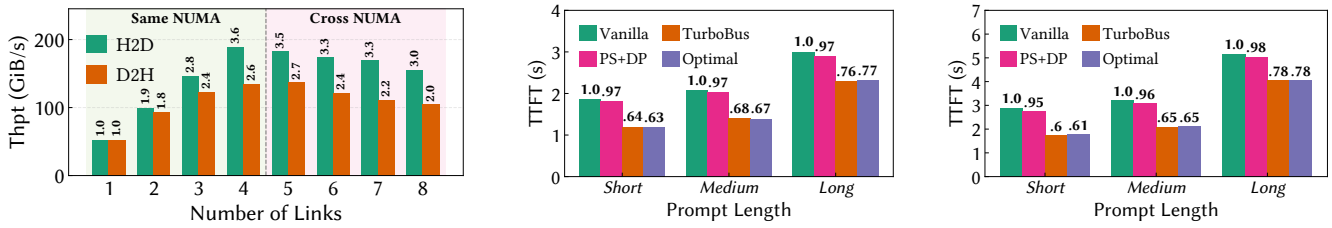
5.2 Micro-Benchmark

Efficacy of pipelining with profiled block size. Fig. 8 (c) shows the bandwidth achieved when transferring 2 GB of data with varying block sizes on the V100 testbed. The arch-shaped curve confirms that both too-small blocks (excessive kernel launches and synchronization) and too-large blocks (warm-up/cool-down stalls that reduce NVLink-PCIe overlap) degrade throughput. The profiled 16 MB block size achieves near-optimal bandwidth, validating our design choice (§3.2.1). To isolate the relay overhead and show how it scales with PCIe speed, we repeat this sweep on the RTX 3090 (PCIe 4.0) and H800 (PCIe 5.0) (Fig. 8 (d) and (e)), using a larger 8 GB transfer on the H800; the dashed lines mark twice the measured single-link PCIe bandwidth, the ideal for pooling two links. At the profiled block size, relay throughput stays close to this two-link ideal on all three platforms, within about 5% for H2D and 6% for D2H, so the relay overhead stays small rather than growing on faster PCIe. This holds because the profiler picks a larger block on faster hardware (16 MB on V100 and RTX 3090, 32 MB on H800), amortizing the fixed per-block overhead that would otherwise consume a larger fraction of the block-transfer time. Because the large-block end of each sweep is effectively store-and-forward, the drop from the peak to it also quantifies the benefit of pipelining.

Block-level path allocation. We assess the performance gains from our path allocation strategy by measuring the throughput of 100 generated transfers with sizes ranging from 30 to 256 MB. Fig. 8 (a) compares our block-level path allocation (§3.2.2) against a coarse-grained, transfer-level baseline. The coarse-grained approach, which locks paths to a single transfer until completion, leaves paths idle during the final pipeline stages. In contrast, our allocator dynamically reassigns idle paths to pending blocks of concurrent transfers, improving throughput by up to 20%.

Concurrent bidirectional transfers. Fig. 8 (b) validates our concurrent bidirectional transfer policy (§3.3.1). This experiment measures the mean throughput in each direction when executing simultaneous H2D and D2H operations (1 GB each) on a shared NVLink path (Fig. 6 (a)). By permitting both directions to share the same NVLink, TurboBus achieves substantially higher aggregate throughput than serializing the two directions, with either H2D or D2H prioritized (i.e., H2D-P and D2H-P). We attribute this gain to two factors: NVLink's bandwidth headroom, which accommodates bidirectional traffic without becoming the bottleneck, and synchronized pipelining (§3.3.1), which avoids CUDA runtime interference when both directions run concurrently. Together, they ensure that PCIe links remain fully saturated.

Impact of scheduling policies. Finally, we evaluate the request prioritization policies (§3.3.2), focusing on chunking and token-based concurrency control. Fig. 8 (f) and (g) show results for D2H and H2D, respectively. Each figure contains two bar groups, "Chunking" and "Token", comparing completion time without (left bar) and with (right bar) the mechanism enabled. For the "Chunking" group, we issue two concurrent transfers of 1 GB and 32 MB. Without chunking, strict FCFS causes head-of-line blocking where the large transfer delays the small one. With chunking enabled, small transfers interleave with chunks of large ones, reducing completion time by 27%. For the "Token" group, we issue two concurrent 1 GB transfers. Without the token, the two large transfers share all paths equally, so each runs at half bandwidth and both finish



(a) Relay scaling on H800 (b) Llama-3-8B (c) Llama-2-13B

Figure 9: (a) Relay scaling on an eight-GPU H800 server with two NUMA nodes (four GPUs each) and pinned memory on the local node: throughput scales near-linearly within the local NUMA node, while equal striping onto the remote node crosses UPI and lowers aggregate throughput. (b–c) TurboBus achieves near-optimal performance for on-demand model loading across prompt lengths.

late. With token-based control, large transfers are serialized so only one proceeds at a time, reducing completion time by 15%. Together, chunking bounds blocking for small transfers and token-gating serializes large transfers to avoid mutual interference, cutting completion time in both cases.

Scaling and NUMA sensitivity. Fig. 9 (a) scales the relay from one to eight PCIe links on an eight-GPU H800 server whose two NUMA nodes hold four GPUs each, with pinned host memory on the local node. The single-link bar is a direct transfer over the owner's own PCIe link, and each added link is one relay's contribution, so the figure also separates the direct and relay contributions. Within the local NUMA node, scaling is near-linear for H2D, reaching 3.6× a single link at four links; D2H scales more modestly to 2.6×. Because the allocator stripes blocks equally across all links it uses, beyond four links the extra links fall on the remote node and must cross UPI to reach pinned memory, stalling the synchronized request, so aggregate H2D throughput declines to 2.9× at eight links. Conversely, on our V100 and RTX 3090 testbeds the local and relay GPUs sit on different NUMA nodes, yet two-link pooling still approaches the two-link ideal (Fig. 8 (c–d)), since the small demand fits within the cross-socket path's capacity. The condition for benefit is therefore not NUMA locality itself, but whether the pooled demand stays within the capacity of the host-side path (§7). A NUMA-aware allocator that prioritizes local links, and borrows remote ones only when they add net bandwidth, would extend the benefit at larger scale; we leave this to future work.

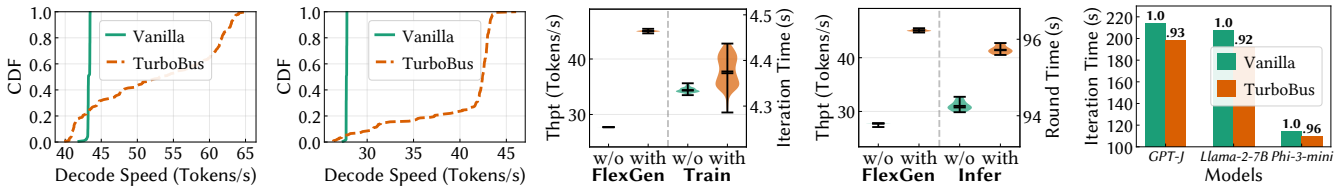
5.3 End-to-End Experiment

5.3.1 On-Demand Model Loading and Serving. We evaluate TurboBus for on-demand model loading in two scenarios: a realistic serverless emulation to assess statistical performance under load, and a controlled isolation study to determine peak theoretical efficiency. We focus on the distribution of Time-to-First-Token (TTFT), a critical Service Level Objective (SLO) metric for serverless inference [21]. We compare three approaches: native GPU–host transfer (Vanilla), PipeSwitch+DeepPlan (PS+DP), and TurboBus. PS+DP combines PipeSwitch's layer-wise transfer-compute pipelining [5] with DeepPlan's multi-GPU relay mechanism [27], representing the state-of-the-art for *intra-job* PCIe optimization. This baseline can aggregate PCIe bandwidth across GPUs belonging to the *same* job but cannot leverage idle links from co-located jobs, precisely the limitation TurboBus addresses.

Performance under serverless load. To evaluate TurboBus in a realistic multi-tenant environment, we emulate a serverless serving workload where models are loaded on-demand and evicted after a short session. We generate 100 inference requests on each GPU, with request arrivals following a Poisson process ($\lambda = 0.5$ requests/s per GPU) to emulate realistic bursty traffic. Each of the four GPUs runs an independent request stream and thus acts as a separate co-located job; TurboBus pools idle PCIe links across these jobs, which the intra-job baselines cannot. Each request simulates a multi-turn conversation consisting of 8 interaction rounds using the Llama-3-8B model [24], with each round generating 128 output tokens. To capture performance variance across different workloads, we assign each request one of three distinct prompt lengths with equal probability: short (19 tokens), medium (216 tokens), and long (823 tokens), corresponding to Fig. 8 (h), (i), and (j), respectively.

The Cumulative Distribution Function (CDF) of the TTFT is shown in Fig. 8 (h–j). The results indicate that TurboBus reduces the median (P50) TTFT by 17%–25% compared to the baselines across all prompt lengths. PipeSwitch's layer-wise pipelining provides limited benefit here because data transfer, not computation, is the bottleneck; DeepPlan, as an intra-job solution, cannot leverage idle links from co-located jobs. At the extreme tail (P99), TurboBus exhibits slightly higher latency, up to 10% compared to PS+DP, due to occasional resource contention when multiple requests compete for shared relay paths during peak congestion. However, for the vast majority of requests, TurboBus is substantially faster, with the 17%–25% median reduction far outweighing the tail regression. This trade-off is well-suited for throughput-oriented serverless deployments, where median latency dominates SLO compliance [21]. The tail is also bounded by design. The chunking and token-gating of §3.3.2 cap how long a request waits behind others at a single chunk duration, so the regression is bounded blocking rather than unbounded queueing.

Peak performance in isolation. To quantify the maximum potential speedup, we run a controlled experiment without external interference. One GPU runs the application while another serves as a dedicated relay, aggregating two PCIe links. We evaluate Llama-3-8B [24] and Llama-2-13B [38] using the same prompt lengths as the serverless experiment above. We compare results against the analytical upper bound from the Amdahl-style model introduced in §5.1. With this setup, the analytical optimum becomes $T_{\text{opt}} = T_c + T_d/2$; this bound is conservative as noted in §5.1.



(a) OPT-6.7B (b) OPT-13B (c) Training (d) Inference (e) ZeRO-Offload

Figure 10: (a–b) Mutual relaying improves throughput by up to 50%; (c–d) heterogeneous co-location achieves up to 1.6× speedup with less than 1% overhead on co-located jobs; (e) TurboBus speeds up ZeRO-Offload by 4%–7% for memory-constrained training.

As shown in Fig. 9 (b) and (c), TurboBus consistently outperforms both Vanilla and PS+DP. Specifically, TurboBus reduces TTFT by 24%–36% for Llama-3-8B and 22%–40% for Llama-2-13B, achieving up to 40% improvement. Notably, TurboBus achieves performance within 5% of the analytical optimum, validating that our relay mechanism introduces minimal overhead. The results also highlight that TurboBus provides greater relative benefits for shorter prompts. With short prompts, model loading dominates the total latency; as prompt length increases (e.g., to 823 tokens), the computation overhead begins to dilute the loading savings. Nevertheless, even for longer prompts, TurboBus improves TTFT by 22%–24%, demonstrating its versatility across different usage patterns.

5.3.2 KV-Cache-Offloaded Serving. This section evaluates TurboBus on FlexGen [37], focusing on KV cache offloading (model weight offloading is covered above) and its impact on co-located jobs. We use the default FlexGen configuration: batch size 32, input length 512 tokens, and output length 32 tokens. The offload policy keeps model weights on GPU while offloading 90% of KV cache to CPU memory, representing a memory-constrained, throughput-oriented configuration. We compare vanilla FlexGen against FlexGen with TurboBus; PipeSwitch and DeepPlan do not apply here, as they target layer-wise model loading rather than KV cache transfers. Multi-tenant GPU clouds exhibit diverse job mixes, but for relaying they reduce to two cases by whether the relay partner is itself an offloading job. We evaluate both: (1) two offloading jobs relay for each other, and (2) an offloading job borrows idle PCIe from a neighbor running a compute- or memory-bound workload.

Homogeneous co-location. We run two FlexGen instances on two GPUs, matching the number of independent PCIe links in our testbed. Each GPU serves as the other's relay. The experiments use OPT-6.7B and OPT-13B [52], running 100 rounds of offline inference and recording their decoding speed. Fig. 10 (a) and (b) show the distribution of decoding speed. Even though both jobs are PCIe-heavy and share the pooled links, TurboBus outperforms vanilla FlexGen in over 80% of iterations, improving throughput by up to 50%; the remaining cases show at most 3 tokens/s slowdown. This confirms our premise (§2.2.2): even co-located PCIe-heavy jobs can pool bandwidth, since their bursty demand leaves idle windows to borrow, and TurboBus's scheduling shares them fairly so both gain rather than either being starved.

Heterogeneous co-location. We run FlexGen alongside another workload on a neighboring GPU, which also serves as FlexGen's relay. The neighboring workload is either training (compute-bound) or inference (memory-bound), representing a broad spectrum of GPU jobs. Fig. 10 (c) shows co-location with a training job: TurboBus improves FlexGen throughput by up to 1.6× (60% speedup) while introducing less than 1% overhead on the co-located job. Fig. 10 (d) shows similar results for an inference workload, with comparable speedup and negligible overhead on the co-located job.

5.3.3 Memory-Constrained Training. We evaluate ZeRO-Offload with three models (GPT-J [41], Llama-2-7B [38], and Phi-3-mini [3]), representing diverse model families. We run ZeRO-Offload with data parallelism across four GPUs (DP=4): each rank offloads its own optimizer-state partition, and the ranks synchronize gradients via NVLink all-reduce, so relay traffic and D2D collective traffic share the fabric. For each model, we use the largest micro-batch size that fits in GPU memory, and apply gradient accumulation (§2.1.1) to reach an effective batch size of 32. Compared to vanilla ZeRO-Offload, TurboBus speeds up training by 4%–7% as shown in Fig. 10 (e).

The modest improvement, compared to on-demand loading, stems from ZeRO-Offload's workload characteristics. Profiling shows PCIe transfers account for only 22%–30% of iteration time, so even a 2× throughput improvement yields at most 12%–17% end-to-end speedup. Our 4%–7% improvement captures 33%–41% of this analytical maximum. Despite the modest relative gains, the absolute savings remain meaningful: a 5% speedup on a week-long job saves over 8 hours of GPU time.

6 Related Work

We relate TurboBus's *cross-job* PCIe pooling to four areas: *intra-job* transfer optimization (its closest prior work), orthogonal data placement and compression techniques, the offloading workloads it targets, and broader resource-pooling systems.

Intra-job transfer optimization. Several systems accelerate GPU–host transfers but are limited to intra-job scope. DeepPlan [27], FaaSTube [45], and Torpor [51] use NVLink-connected GPUs as relays to aggregate PCIe bandwidth, but support only H2D transfers and cannot relay through another job's GPUs. memHarvester [8] enables multi-path access through a tiered memory hierarchy but relies on page-fault-driven migration suited to workloads with access locality. All four use store-and-forward, completing one link's transfer before starting the next. Orthogonally, PipeSwitch [5] and Mobius [20] overlap transfers with computation via layer-wise pipelining, rather than increasing the usable bandwidth TurboBus provides. None can leverage idle PCIe links from co-located jobs. Pipelined forwarding itself is known: Joshi et al. [28] characterize PCIe-to-NVLink-to-PCIe forwarding over alternative host-memory paths. TurboBus's novelty is the cross-job setting, which requires isolation, bidirectional scheduling, and concurrency control that

single-job designs lack. FastPersist [42] parallelizes GPU-to-SSD transfers across data-parallel replicas, pooling storage I/O rather than the GPU–host PCIe bandwidth TurboBus targets.

Placement and compression. Orthogonal to TurboBus, which increases transfer bandwidth, these approaches reduce the data that must be transferred, either by keeping it resident near the GPU or by compressing it. ServerlessLLM [21] and Aegaeon [48] place models closer to GPUs via hierarchical storage and residency scheduling, respectively. InfiniGen [30] and StellaTrain [31] reduce transfer volume through KV cache approximation and gradient compression, respectively. Optimus [25] shares model weights across functions to avoid redundant loading. These approaches are composable with TurboBus, which accelerates whatever transfers remain.

Offloading workloads. TurboBus benefits workloads exhibiting the individually bottlenecked, collectively underutilized pattern. *On-demand model loading* [5, 21, 27, 48] fetches model weights at inference time, bottlenecking cold-start latency. *KV cache offloading* [7, 22, 29, 32, 37, 53] swaps attention states to host memory, creating bursty PCIe traffic during decoding. *Memory-constrained training* [20, 34, 35, 43] offloads optimizer states or full model states to host memory, making PCIe throughput critical for iteration time. All three are bottlenecked by PCIe transfers, making them natural targets for TurboBus's bandwidth pooling.

Resource pooling. AQUA [40] pools GPU memory; CXL-based disaggregation [47, 54] enables CPU memory pooling; Prism [50] disaggregates CPUs and GPUs for serving. Most relevant is FuseLink [36], which uses contention-aware scheduling to pool inter-server NIC bandwidth via GPU relays. TurboBus applies this principle to intra-node PCIe bandwidth, where tighter latency and bidirectional, cross-job transfers demand new mechanisms.

7 Discussion

Hardware requirements and generalization. TurboBus requires high-bandwidth GPU-to-GPU interconnects (NVLink, NVSwitch) to make relay overhead negligible. On systems with only PCIe peer-to-peer connectivity, relay transfers would contend for the same PCIe bandwidth they aim to pool, negating the benefit. As a rule of thumb, relay is beneficial once GPU-to-GPU bandwidth exceeds the PCIe bandwidth; beyond twice the PCIe bandwidth, even concurrent bidirectional transfers (§3.3.1) incur no NVLink bottleneck. Our evaluation already validates this across three PCIe generations (V100, RTX 3090, and H800; §5.2), spanning both switched-fabric and pairwise NVLink. We expect TurboBus to generalize further, as the requirement holds across datacenter GPUs (V100, A100 [15], H100 [14]) and fabrics (AMD Infinity Fabric [4], UALink [9]), including the newest B200 [16], where the NVLink-to-PCIe gap reaches ~7× (NVLink5 ~900 vs. PCIe Gen6 ~128 GB/s per direction). PCIe also remains the offloading bottleneck on the newest GPUs, since GPU compute and on-package memory bandwidth grow at least as fast, so a GPU's offloading demand keeps saturating its own link. With both the bottleneck and the asymmetry intact, TurboBus's relative improvements should persist.

Host-side bottlenecks differ across platforms. On larger nodes the binding host-side resource is platform-dependent, so more GPUs do not translate to proportionally more pooled bandwidth (Fig. 9 (a)). On an eight-GPU DGX A100 [11, 15], each PCIe switch multiplexes two GPUs onto one Gen4 ×16 uplink to the CPU. The four GPUs in a socket therefore reach host memory through only two uplinks, about ~64 GB/s per direction. This is far below the socket's ~205 GB/s of eight-channel DDR4-3200 bandwidth. Here the PCIe switch fabric, not the memory controller, caps pooling. On a DGX H100 [14], each GPU instead has a dedicated Gen5 ×16 link (~64 GB/s per direction). The four GPUs in a socket can then drive ~256 GB/s per direction. Under combined H2D and D2H traffic this exceeds the socket's ~307 GB/s of DDR5-4800 bandwidth, a half-duplex read+write ceiling. Here the memory controller caps pooling instead.

Integrated CPU–GPU architectures. Tightly integrated designs couple CPU and GPU memory over a fast coherent interconnect, removing PCIe from the GPU–host path. Examples are NVIDIA Grace Hopper/Blackwell (NVLink-C2C, up to 900 GB/s) [12] and AMD MI300A [19]. On such nodes the GPU–host path is no longer PCIe-bottlenecked, so TurboBus's relay offers no benefit; we scope it to PCIe-connected servers. However, two factors keep the approach relevant. First, TurboBus targets the large installed base of PCIe-connected multi-GPU servers (DGX A100/H100 and cloud instances), which remain dominant and will coexist with integrated nodes for years. Second, the bottleneck shifts rather than vanishes. On Grace Hopper the host's memory bandwidth (~500 GB/s LPDDR5X) is below the 900 GB/s C2C link, so a GPU streaming from host memory can be host-bound. Pooling idle host-memory paths across GPUs (e.g., across NUMA domains) follows the same borrow-the-faster-fabric principle, but is a future direction rather than a direct extension. Pinned memory is NUMA-local, so a remote path cannot reach the data at full bandwidth unless the blocks are placed on that domain; our NUMA scaling results show this cost (Fig. 9 (a)). Realizing the principle on such platforms requires memory-placement support that TurboBus does not handle, which we leave to future work.

8 Conclusion

Workloads that offload GPU memory are bottlenecked by PCIe bandwidth yet leave most link capacity idle, a paradox rooted in per-GPU link ownership. TurboBus resolves this problem by pooling PCIe bandwidth across co-located jobs via NVLink relay, preserving process isolation, requiring only minimal code changes, and providing explicit flow scheduling. Our evaluation demonstrates up to 40% TTFT reduction for on-demand model loading (within 5% of the analytical optimum), up to 1.6× throughput for KV-cache-offloaded inference, and up to 7% speedup for memory-constrained training, while imposing less than 1% overhead on co-located workloads.

Ethics statement. *This work does not raise any ethical issues.*

9 Acknowledgement

We thank the anonymous SIGCOMM reviewers and our shepherd for their insightful comments. This work is supported in part by the Hong Kong RGC TRS T41-603/20R and TACC [49]. Kai Chen is the corresponding author.

References

[1] 2026. Azure Dedicated Host. Microsoft Azure Documentation. https://azure.microsoft.com/en-us/products/virtual-machines/dedicated-host/ Accessed 2026-01-26.

[2] 2026. Dedicated Instances. Amazon Web Services Documentation. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/dedicated-instance.html Accessed 2026-01-26.

[3] Marah Abdin et al. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. *arXiv preprint arXiv:2404.14219* (2024). https://arxiv.org/abs/2404.14219

[4] Advanced Micro Devices, Inc. 2020. *AMD Infinity Fabric™ Link Installation Guide*. Technical Report. AMD. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/other/56978.pdf

[5] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 499–514.

[6] Mike W. Blasgen, Jim Gray, Michael F. Mitoma, and Thomas G. Price. 1979. The Convoy Phenomenon. *ACM SIGOPS Oper. Syst. Rev.* 13, 2 (1979), 20–25. https://doi.org/10.1145/850657.850659

[7] Hongtao Chen, Weiyu Xie, Boxin Zhang, Jingqi Tang, Jiahao Wang, Jianwei Dong, Shaoyuan Chen, Ziwei Yuan, Chen Lin, Chengyu Qiu, Yuening Zhu, Qingliang Ou, Jiaqi Liao, Xianglin Chen, Zhiyuan Ai, Yongwei Wu, and Mingxing Zhang. 2025. KTransformers: Unleashing the Full Potential of CPU/GPU Hybrid Inference for MoE Models. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. ACM, Lotte Hotel World Seoul Republic of Korea, 1014–1029. https://doi.org/10.1145/3731569.3764843

[8] Sangjin Choi, Taeksoo Kim, Jinwoo Jeong, Rachata Ausavarungnirun, Myeongjae Jeon, Youngjin Kwon, and Jeongseob Ahn. 2022. Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 2022), Carlsbad, CA, USA, July 11–13, 2022*. USENIX Association, 625–638. https://www.usenix.org/conference/atc22/presentation/choi-sangjin

[9] UALink Consortium. 2025. *UALink Technical Overview*. https://nand-research.com/research-note-ualink-alliance-accelerator-interconnect-specification/ Accessed: 2025-12-19.

[10] NVIDIA Corporation. 2017. *NVIDIA DGX-1 With Tesla V100 System Architecture*. https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf Accessed: 2025-12-19.

[11] NVIDIA Corporation. 2020. *NVIDIA DGX A100 System Architecture*. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/dgx-a100/dgxa100-system-architecture-white-paper.pdf White Paper WP-10083-001; Accessed: 2026-06-04.

[12] NVIDIA Corporation. 2023. *NVIDIA Grace Hopper Superchip*. https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/ Accessed: 2026-06-03.

[13] NVIDIA Corporation. 2024. *CUDA C++ Best Practices Guide*. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html Accessed: 2026-06-26.

[14] NVIDIA Corporation. 2024. *Introduction to NVIDIA DGX H100/H200 Systems — NVIDIA DGX H100/H200 User Guide*. https://docs.nvidia.com/dgx/dgxh100-user-guide/introduction-to-dgxh100.html Accessed: 2025-12-19.

[15] NVIDIA Corporation. 2024. *Introduction to the NVIDIA DGX A100 System — NVIDIA DGX A100 User Guide*. https://docs.nvidia.com/dgx/dgxa100-user-guide/introduction-to-dgxa100.html Accessed: 2025-12-19.

[16] NVIDIA Corporation. 2024. *NVIDIA GB200 NVL72*. https://www.nvidia.com/en-us/data-center/gb200-nvl72/ Accessed: 2026-01-28.

[17] NVIDIA Corporation. 2025. *Multi-Process Service (MPS)*. https://docs.nvidia.com/deploy/mps/index.html Accessed: 2026-02-04.

[18] NVIDIA Corporation. 2025. *NVIDIA Management Library (NVML)*. https://developer.nvidia.com/management-library-nvml Accessed: 2025-01-22.

[19] Advanced Micro Devices. 2024. *AMD Instinct MI300A APU*. https://www.amd.com/en/products/accelerators/instinct/mi300/mi300a.html Accessed: 2026-06-15.

[20] Yangyang Feng, Minhui Xie, Zijie Tian, Shuo Wang, Youyou Lu, and Jiwu Shu. 2023. Mobius: Fine Tuning Large-Scale Models on Commodity GPU Servers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 489–501. https://doi.org/10.1145/3575693.3575703

[21] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 135–153.

[22] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 111–126.

[23] Georgi Gerganov and contributors. 2023. *llama.cpp*. https://github.com/ggml-org/llama.cpp LLM inference in C/C++.

[24] Aaron Grattafiori et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024). https://arxiv.org/abs/2407.21783

[25] Zicong Hong, Jian Lin, Song Guo, Sifu Luo, Wuhui Chen, Roger Wattenhofer, and Yue Yu. 2024. Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1039–1053. https://doi.org/10.1145/3627703.3629567

[26] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*. USENIX Association, 947–960. https://www.usenix.org/conference/atc19/presentation/jeon

[27] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. 2023. Fast and Efficient Model Serving Using Multi-GPUs with Direct-Host-Access. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 249–265. https://doi.org/10.1145/3552326.3567508

[28] Raj Joshi, Saksham Agarwal, ChonLam Lao, and Minlan Yu. 2025. Your Network Doesn't End at the NIC: A Case for Unifying the Inter-Host and Intra-Host Networks in (AI) Datacenters. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks (HotNets '25)*. ACM, 280–288. https://doi.org/10.1145/3772356.3772415

[29] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[30] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 155–172.

[31] Hwijoon Lim, Juncheol Ye, Sangeetha Abdu Jyothi, and Dongsu Han. 2024. Accelerating Model Training in Multi-cluster Environments with Consumer-grade GPUs. In *Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 707–720. https://doi.org/10.1145/3651890.3672228

[32] Yuhan Liu, Jiayi Yao, Yihua Cheng, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. 2025. LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference. *arXiv preprint arXiv:2510.09665* (2025). https://arxiv.org/abs/2510.09665

[33] Meta AI. 2025. Llama — Industry Leading, Open-Source AI. https://www.llama.com/.

[34] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3458817.3476205

[35] 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.

[36] Zhenghang Ren, Yuxuan Li, Zilong Wang, Xinyang Huang, Wenxue Li, Kaiqiang Xu, Xudong Liao, Yijun Sun, Bowen Liu, Han Tian, Junxue Zhang, Mingfei Wang, Zhizhen Zhong, Guyue Liu, Ying Zhang, and Kai Chen. 2025. Enabling Efficient GPU Communication over Multiple NICs with FuseLink. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 91–108.

[37] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *International Conference on Machine Learning (ICML 2023)*. PMLR, 31094–31116. https://proceedings.mlr.press/v202/sheng23a.html

[38] Hugo Touvron et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288* (2023). https://arxiv.org/abs/2307.09288

[39] Ubicloud Team. 2025. Virtualizing NVIDIA HGX B200 GPUs with Open Source. Ubicloud Blog. https://www.ubicloud.com/blog/virtualizing-nvidia-hgx-b200-gpus-with-open-source Accessed 2026-01-26.

[40] Abhishek Vijaya Kumar, Gianni Antichi, and Rachee Singh. 2025. AQUA: Network-Accelerated Memory Offloading for LLMs in Scale-Up GPU Domains. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, Rotterdam Netherlands, 48–62. https://doi.org/10.1145/3676641.3715983

[41] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.

[42] Guanhua Wang, Olatunji Ruwase, Bing Xie, and Yuxiong He. 2024. FastPersist: Accelerating Model Checkpointing in Deep Learning. arXiv:2406.13768 [cs]

[43] Jiali Wang, Yankui Wang, Mingcong Han, and Rong Chen. 2025. Colocating ML Inference and Training with Fast GPU Memory Handover. In *2025 USENIX Annual Technical Conference (USENIX ATC '25)*. USENIX Association. https://www.usenix.org/conference/atc25/presentation/wang-jiali

[44] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 364–381. https://doi.org/10.1145/3600006.3613145

[45] Hao Wu, Junxiao Deng, Minchen Yu, Yue Yu, Yaochen Liu, Hao Fan, Song Wu, and Wei Wang. 2024. FaaSTube: Optimizing GPU-oriented Data Transfer for Serverless Computing. https://doi.org/10.48550/arXiv.2411.01830 arXiv:2411.01830

[46] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. 2025. Greyhound: Hunting Fail-Slows in Hybrid-Parallel Training at Scale. In *2025 USENIX Annual Technical Conference (USENIX ATC 2025), Boston, MA, USA, July 2025*. USENIX Association, 731–747. https://www.usenix.org/conference/atc25/presentation/wu-tianyuan

[47] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2024), Santa Clara, CA, USA, July 10–12, 2024*. USENIX Association, 19–35. https://www.usenix.org/conference/osdi24/presentation/xiang

[48] Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin, and Jingren Zhou. 2025. Aegaeon: Effective GPU Pooling for Concurrent LLM Serving on the Market. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 1030–1045. https://doi.org/10.1145/3731569.3764815

[49] Kaiqiang Xu, Decang Sun, Hao Wang, Zhenghang Ren, Xinchen Wan, Xudong Liao, Zilong Wang, Junxue Zhang, and Kai Chen. 2025. Design and Operation of Shared Machine Learning Clusters on Campus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 295–310. https://doi.org/10.1145/3669940.3707266

[50] Lingyun Yang, Yongchen Wang, Yinghao Yu, Qizhen Weng, Jianbo Dong, Kan Liu, Chi Zhang, Yanyi Zi, Hao Li, Zechao Zhang, Nan Wang, Yu Dong, Menglei Zheng, Lanlan Xi, Xiaowei Lu, Liang Ye, Guodong Yang, Binzhang Fu, Tao Lan, Liping Zhang, Lin Qu, and Wei Wang. 2025. GPU-Disaggregated Serving for Deep Learning Recommendation Models at Scale. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 2025), Philadelphia, PA, USA, April 28–30, 2025*. USENIX Association, 847–863. https://www.usenix.org/conference/nsdi25/presentation/yang

[51] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, Haoran Yang, and Yu Ding. 2025. Torpor: GPU-Enabled Serverless Computing for Low-Latency, Resource-Efficient Inference. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 597–612.

[52] Susan Zhang et al. 2022. OPT: Open Pre-trained Transformer Language Models. *arXiv preprint arXiv:2205.01068* (2022). https://arxiv.org/abs/2205.01068

[53] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*. https://arxiv.org/abs/2312.07104

[54] Yuhong Zhong, Daniel S. Berger, Carl A. Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2024), Santa Clara, CA, USA, July 10–12, 2024*. USENIX Association, 37–56. https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong