Breaking the Tradeoff: Elastic and Isolated GPU Sharing with Ghost

Yicheng Liu[*,1] Yifan Qiao[*†,2] Tian Xia[2] Yi Xu[2] Tony Hong[2] Shuo Yang[2] Yilong Zhao[2]
Jiarong Xing[3] Harry Xu[1] Ion Stoica[2] Joseph E. Gonzalez[2] Sam Kumar[1]

UCLA[1] UC Berkeley[2] Rice University[3]

Abstract

GPUs are expensive resources but often underutilized. This inefficiency is exacerbated by the increasing diversity of AI workloads, where small or specialized models often fail to saturate high-capacity devices, necessitating GPU sharing. Existing sharing approaches force a hard tradeoff. They either partition hardware to provide strong isolation but sacrifice elasticity, or employ application-level resource sharing to provide flexibility at the expense of memory isolation or fault containment. We present Ghost, an OS-level GPU virtualization layer integrated directly into the open-source GPU driver. Ghost uses three mechanisms to break this hard tradeoff. (1) a GPU container abstraction with `cgroup`-like APIs for compute and memory control, (2) fine-grained per-container memory accounting and proactive swapping via extended GPU page tables and fault handlers, and (3) privileged hardware-level scheduling and preemption for dynamic compute resource management. Fully compatible with existing applications, Ghost improves throughput by up to 2× and reduces latency by up to 58× across two representative AI frameworks.

1 Introduction

GPUs are costly but often underutilized in production. Recent hardware trends show rapid growth in HBM capacity, bandwidth, and compute FLOPs; for example, the NVIDIA B200 features 180GB of HBM with 8TB/s bandwidth, and 72 PFLOPS in FP8 [49], at a cost of approximately $500k. Yet, modern AI workloads are increasingly diverse. Many specialized models are too small or lack sufficient intensity to saturate such devices. As a result, major datacenters report that GPU utilization typically remains below 30% for compute and below 60% for memory [13, 70].

One natural approach to improve utilization is providing GPU virtualization and GPU sharing across multiple applications [13, 22, 37, 42, 46, 48, 60, 80]. This is particularly relevant for modern compound AI systems [50, 63, 64, 73, 78] and multi-agent frameworks [1, 15, 19, 54]. These systems consist of multiple distinct components that vary in their resource requirements. For instance, GPT-oss-120B [51] consumes ~65GB HBM, whereas smaller models like GPT-oss-20B [51] or Qwen3-Omni talker [73] require only 10–13GB. A single high-end GPU like H100 or B200 can theoretically colocate these models to achieve much higher density.

	HW. Partitioning	MPS (-based)	Kernel Split (Tally/LithOS)	Kernel Sched. (GPreempt/XSched)	Mem. Intercept (TGS)	Ours
Transparency	✓	✓	✗	✓	✓	✓
Isolation	✓	✗	●	✓	✗	✓
Elasticity	✗	✗	✗	✗	✓	✓

Table 1: An overview of representative GPU sharing solutions.

Effective sharing requires *elasticity*: the ability to dynamically reassign underutilized resources to other applications. There exists a large body of *application-level solutions*, including NVIDIA MPS [42] and recent research systems [13, 18, 22, 37, 60, 77, 80], which provide elasticity by intercepting or modifying kernel launch calls to reorder execution. Such solutions suffer from *noisy neighbor* effects and failure propagation across compute and memory. To illustrate, consider a GPU-memory constrained environment where the aggregate memory required by multiple applications running on the same GPU exceed its memory capacity: a single process allocating excessive memory can trigger CUDA out-of-memory errors that crash co-located applications. Even if the system supports memory oversubscription [5, 39], contention for GPU memory can cause severe performance degradation. For instance, our analysis shows that running a diffusion model concurrently with an LLM can increase the LLM's P99 latency by more than 10× due to memory contention (§2.3).

Therefore, a practical sharing system must pair elasticity with *strong isolation*, encompassing both fault and performance isolation, so that one application's behavior does not crash or significantly delay co-located workloads. Strong isolation can be obtained by using *hardware-level partitioning mechanisms* provided by GPU vendors, such as NVIDIA MIG [46] and vGPU [48], which statically divide GPU compute and memory resources. However, reconfiguring these partitions typically requires a full GPU reset and termination of running applications, making them unsuitable for dynamic workloads.

Although there is a strong need for a GPU system that simultaneously provides elasticity and isolation for mod-

[*] Yicheng Liu and Yifan Qiao contributed equally.
Corresponding authors: Harry Xu, Ion Stoica, Joeseph E. Gonzales, and Sam Kumar.

ern AI workloads, building such a system is fundamentally challenging. Elasticity is a *high-level* property tied to *application behavior*—an application's characteristics determine its resource needs and whether unused resources can be harvested by others. In contrast, isolation is a *low-level* property that relies on hardware mechanisms for precise memory and compute accounting, as well as controlled page table access.

As GPU hardware advances rapidly, the tension between these two levels intensifies, driven by the widening gap between *increasingly powerful low-level GPU capabilities* and the *outdated high-level abstractions exposed to applications*. On one hand, modern GPUs have evolved into managed devices with rich control mechanisms, including preemption support, scheduling hooks, and page tables. On the other hand, they still lack a *container-like* abstraction, leaving GPU applications without the hardware-enforced isolation guarantees that have long been available to CPU workloads. As a result, existing sharing systems face a difficult trade-off: treat GPUs as black boxes and implement elasticity at the application level without strong isolation, or rely on hardware partitioning with strong isolation but little elasticity.

Insight. Breaking this tradeoff requires a middle-layer solution that provides a stronger abstraction—one that translates application semantics into hardware-enforceable policies while exposing the underlying hardware capabilities to applications. This observation directs our attention to the GPU driver in the OS kernel. Positioned at the middle of the stack, the driver has access to low-level hardware mechanisms, while also serving as the primary interface to applications. Recently, major GPU vendors including NVIDIA [45] and AMD [7] have begun open-sourcing their GPU drivers, exposing, for the first time, interfaces to low-level GPU mechanisms for virtual memory management and scheduling. This development creates an opportunity to elevate the GPU driver into a middle-layer "resource hypervisor" that can flexibly virtualize and multiplex diverse GPU resources.

Building on this insight, we develop Ghost, a novel GPU virtualization system integrated into the open-source GPU kernel driver [45]. Ghost fundamentally breaks the existing tradeoff between elasticity and isolation by introducing a *GPU container* abstraction that offers `cgroup`-like interfaces for specifying compute and memory quotas. Due to its unique middle-layer position, Ghost enforces these limits with low overhead through low-level hardware mechanisms (*e.g.*, preemption, scheduling hooks, and page fault handling), while dynamically reclaiming and redistributing idle resources. Unlike application-level approaches, Ghost runs with OS kernel privileges, which enables precise memory accounting, enforced memory residency, and fault containment. In contrast to hardware partitioning, Ghost achieves elasticity by continuously monitoring GPU state and adapting resource allocations within the driver, without requiring GPU resets or interrupting running applications. Ghost provides baseline elasticity and isolation entirely *transparently*, requiring no source code modifications. Additionally, it exposes an OS-level signal as an option that allows aware applications to cooperatively achieve even better performance.

Challenges. Although the open GPU driver exposes low-level mechanisms, building a purely software solution that simultaneously provides elasticity and isolation remains challenging for three reasons.

First, enabling elasticity and isolation requires page-level control over GPU memory, but it is hard for the GPU driver to gain such control because the driver typically cannot access page tables or observe page faults; such functionality is restricted to proprietary firmware that runs on the GPU device, not in the driver. However, we observe that the GPU driver can monitor page faults and update mappings for memory allocated via Unified Virtual Memory (UVM) [5, 39]. We therefore intercept memory allocation calls and redirect them to UVM, enabling page fault observability. With all allocations routed through UVM, the host OS driver can attribute each newly allocated physical page to its container during fault handling. This provides a clean mechanism for tracking precise memory usage and enforcing capacity limits through demand paging (with 1% overhead, which is negligible).

Second, it is hard to schedule GPU kernels efficiently across applications due to the high frequency of kernel executions (often complete in <100µs). Using a classic OS scheduler could trigger excessive remote procedure calls (RPCs) to the GPU across PCIe. To address this challenge, Ghost has the driver and GPU hardware perform scheduling collaboratively. On one hand, we let the GPU round-robin scheduler (§3.2) handle low-level, high-frequency decisions to avoid excessive overhead. On the other hand, we uncover additional RPC interfaces to control the GPU firmware from the GPU driver, enabling high-level control such as allocating the timeslices and triggering preemption. This hybrid approach allows our scheduling framework to inject scheduling decisions in an efficient way only when necessary, which minimizes PCIe overhead and outperforms state-of-the-art schedulers (§4.3).

Third, enforcing memory limits inevitably involves handling oversubscription, meaning that data from applications exceeding their memory budgets must be swapped out. However, naïvely swapping data without awareness of memory types or semantics (*e.g.*, KV cache is more amenable to swapping than model weights in an inference engine) can lead to excessive swapping and severe system thrashing. To address this, Ghost draws inspiration from OS interrupt handling by issuing swap signals to applications. Applications can optionally register a handler—separate from their core logic—that identifies memory regions suitable for swapping, while Ghost performs the actual data movement. This design leverages application semantics to achieve better performance. If no handler is provided, Ghost falls back to generic swapping, preserving transparency at the cost of efficiency.

We have implemented Ghost for NVIDIA GPUs and evaluated it on an A100 GPU using two representative AI workloads. Compared to the state of the art, Ghost effectively isolates colocated applications, i.e., reducing latency-sensitive applications' latency by up to 58× while improving throughput-oriented applications' throughput by up to 2×.

2 Background and Motivation

2.1 GPU Utilization Challenges

Various cloud providers, including Microsoft [27], Meta [13], and Alibaba [70], report low GPU utilization. In an Alibaba production cluster [70], nearly half of GPU memory remains stranded, and GPU compute utilization stays below 25%[1] for most of the time. Compute is underutilized because inference demand is bursty—request rates can fluctuate by more than 3× within tens of seconds [68]—and training is punctuated by communication stalls [28] and I/O blocking [65], leaving recurrent idle periods that static reservation cannot reclaim.

Memory is underutilized because providers typically provision GPUs for worst-case peak usage to prevent OOM crashes, often assigning an entire high-end GPU to a single model that consumes only a fraction of its capacity. Moreover, modern AI systems frequently compose heterogeneous models—from trillion-parameter LLMs [14, 34, 62] to specialized models for OCR [69] and image generation [57]—into multi-stage pipelines, further exacerbating fragmentation when each stage is statically allocated.

2.2 Existing GPU Sharing Mechanisms

Existing GPU sharing mechanisms mainly aim to improve utilization but often compromise isolation, elasticity, or both (Table 1). Hardware partitioning (NVIDIA MIG [46], vGPU [48]) offers strong isolation but are inelastic—resizing requires a GPU reset and termination of all applications.

Software-based approaches trade isolation for flexibility. MPS [42] allows concurrent kernels but lacks memory and fault isolation. Tally [81] and LithOS [13] split kernels and run them in shared processes, so a fault in one can propagate to others. GPreempt [16] and XSched [59] employ kernel-level preemption for time-sharing and fault isolation but neglect memory as a shared resource. TGS [71] enables memory oversubscription via host paging but offers no guarantee over per-process memory usage or placement, leading to unpredictable stalls under contention (see §2.3).

2.3 Motivational Performance Study

We co-locate latency-sensitive LLM inference (Llama 3.1 8B on vLLM [33]) with best-effort image generation (Stable Diffusion 3.5 Medium [57]) on an A100-40G GPU, mimicking real-world multimodal pipelines (*e.g.*, GPT-4o).

When both workloads fit in memory (Figure 1a), the default driver's time-sharing inflates vLLM's P99 TTFT and ITL[2] by 10.4× and 2.7×. XSched [59] preserves vLLM latency but reduces diffusion throughput to 23.4% of ideal, even though vLLM uses only 61.2% of SM time.

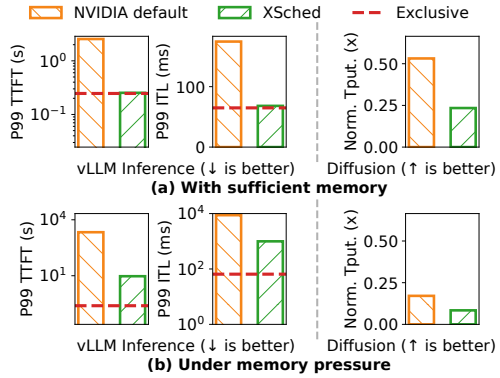


Figure 1: Performance of colocated vLLM and diffusion on an A100-40G GPU. (a) With sufficient memory. (b) Under memory pressure.

Under memory pressure—when vLLM reserves 36GB for KV caches and the combined footprint (56GB) exceeds GPU capacity—XSched with TGS [71] for paging degrades sharply: vLLM's P99 TTFT and ITL inflate to 37× and 15× of ideal, and diffusion throughput falls to 8.4% (Figure 1b). Three root causes drive this degradation:

- **Lack of memory isolation.** Without per-process memory quotas, co-located applications compete for GPU memory and trigger unpredictable latency spikes via demand paging (Figure 2). Enforcing capacity isolation requires control over device physical memory and page residency, which is unattainable in user space.
- **Inefficient memory paging.** The effective swap bandwidth is only 3.4 GB/s (21% of PCIe capacity) because the driver's eviction logic blocks the fault-in path and frequent small control messages prevent bus saturation (Figure 3). The CPU incurs 3× overhead on fault handling, and blocking I/O expands diffusion execution time by 11×.
- **Inefficient compute scheduling.** User-space scheduling (XSched) must insert CUDA events and perform IPC to coordinate processes, achieving only 86.0% compute utilization versus 99.8% for the driver's native hardware-assisted context switching.

Takeaway. User-space GPU sharing systems cannot enforce process-level memory quotas or efficiently schedule without privileged access to GPU page tables and kernel execution status. Efficient resource management requires lower-level hardware observability and control.

3 Dissecting GPU Resource Management

We ground the design of Ghost in a detailed analysis of the open GPU driver [45]. Specifically, we studied the driver's source code to understand its internal architecture and interfaces, and we instrumented its compute and memory paths

[1] Measured as the percentage of time when SMs are activated.

[2] TTFT: *time to first token*; ITL: *inter-token latency*.

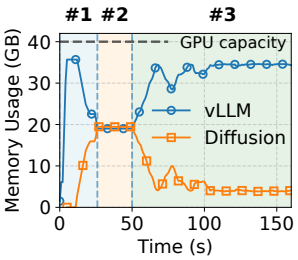


Figure 2: Memory contention visualization when total memory demand exceeds GPU capacity.

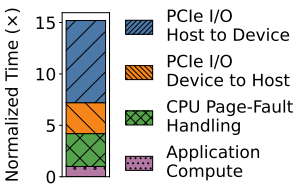


Figure 3: Diffusion time breakdown under memory pressure, normalized to it running with sufficient memory.

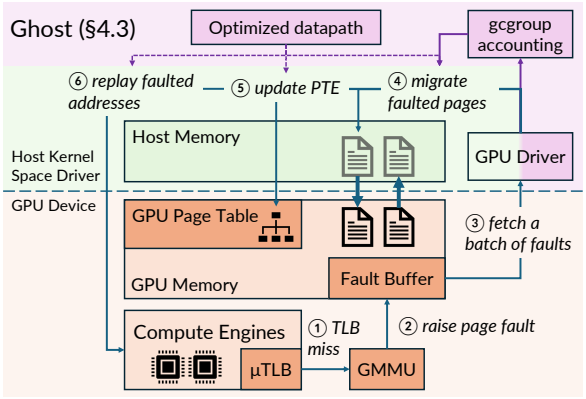


Figure 4: GPU hardware/software architecture for virtual memory management. The GPU MMU translates virtual addresses, and major page faults are handled by the driver in the host OS kernel.

to uncover undocumented hardware mechanisms. The resulting insights shaped our design decisions in §4.

3.1 GPU Virtual Memory

Modern GPUs support virtual memory in a manner analogous to CPUs. Beginning with NVIDIA's Pascal [40] and AMD's Vega [4] architectures, each process executes within its own isolated virtual address space with a dedicated page table. The software stack is divided into two layers: the userspace CUDA runtime, which manages allocations, and the host-side kernel driver, which configures the GPU's MMU and handles page faults.

The NVIDIA GPU driver supports two distinct memory management models. To maintain compatibility with older GPUs, standard allocations using `cudaMalloc` pre-map physical pages immediately, eliminating the possibility of page faults during execution. However, post-Pascal NVIDIA GPUs also support Unified Virtual Memory (UVM) [5, 39], which allocates physical pages on demand and allows page faults and memory overcommitment. UVM presents a unified address space shared between the host and the device and uses page faults to migrate data on demand, so the GPU can overcommit virtual memory and page out excess device-resident data to host memory.

As shown in Figure 4, GPU virtual memory differs from CPU memory management in two key ways. First, GPU page faults are raised on the device but handled on the host. GPU kernels access virtual addresses; the GPU memory management unit (GMMU) translates them and per-SM micro TLBs (μTLBs) cache translations. When an SM core touches a non-resident page, the access triggers a μTLB miss and the GMMU records a page fault. Faults from many SMs are buffered on the device; the GPU then signals the host driver via an interrupt, and the driver fetches the entire batch of fault records over PCIe. On the CPU, the UVM module in the GPU driver decides which pages to bring into device memory and which resident pages to evict, issues DMA transfers, updates page tables, and instructs the GPU to replay the faulting instructions. When device memory is full, the driver evicts pages back to host memory, including data transfers, page-table updates, TLB shootdowns, and bookkeeping.

Second, GPU page faults are reported in batches rather than individually. Raising each fault directly into the OS like a CPU would incur prohibitive PCIe overhead. Instead, the GPU accumulates many faults into a single batch for host-side processing. The GPU's high thread parallelism allows it to tolerate long fault-handling latency by scheduling other warps; the design therefore favors high-throughput, batched handling over low per-fault latency.

In summary, while conceptually similar to CPU paging, GPU fault handling operates at batch granularity and involves multiple PCIe round-trips and synchronous driver work, which can amplify interference under heavy load.

Design implication M1: host-handled GPU faults enable memory isolation. Because faults are handled on the host, the driver can see the PTE content and physical page allocation, and hence it can keep track of the physical memory usage of each process, classify page faults by process, and apply different policies to different tenants.

Design implication M2: efficient paging must minimize PCIe costs. Although the GPU hardware reports faults in batches, the driver still handles them individually via fine-grained RPCs to the device, including updating PTEs and initiating migration I/O. Frequent RPCs incur high CPU and PCIe overheads and throttle effective paging throughput (3.4GB/s as measured in §2.3). Moreover, this inefficiency will become increasingly critical as hardware bandwidth grows (*e.g.*, NVLink [47]).

3.2 GPU Compute Scheduling

GPU compute scheduling is split across the host driver, device firmware, and hardware. A GPU is a massively parallel processor composed of multiple *Streaming Multiprocessors (SMs)*, which execute threads grouped into *warps*. To utilize this hardware, applications launch compiled functions called *kernels* into *streams* (ordered queues of operations).

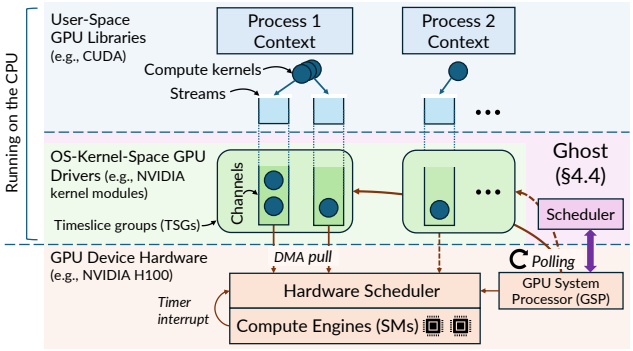


Figure 5: GPU compute scheduling hardware/software architecture.

Unlike scheduling on CPUs where the OS can issue privileged instructions directly to the CPU hardware, GPU compute scheduling is split among the host-side driver, closed-source device firmware running on the GPU system processor (GSP), and the GPU hardware scheduler [10].

As Figure 5 illustrates, the driver implements high-level CUDA abstractions and runs in the host OS kernel space to configure GPU hardware. It creates a *channel* as a DMA push buffer for each CUDA stream, so the process can submit kernels directly to the GPU hardware. It further groups channels of each process into a *time-slice group* (TSG) and organizes all TSGs in a global *runlist* to multiplex the GPU across processes and their TSGs.

The driver does not schedule individual TSGs or kernels. Instead, it offloads TSG scheduling to GSP and kernel scheduling to the GPU hardware scheduler, respectively, to minimize the kernel scheduling overhead. GSP receives the global runlist via remote procedure calls (RPCs) from the driver, and it configures the hardware scheduler to round robin across TSGs given their *timeslices* and pull their kernels for actual scheduling onto SMs.

When a TSG exhausts its timeslice, a hardware timer interrupt fires and triggers a hardware context switch similar to CPUs. In each interrupt, the hardware saves and restores execution states, such as registers, shared memory, and program counters, to device memory. Each such switch typically incurs a latency of tens of microseconds[3] [16].

While this mechanism enables concurrent execution, it lacks explicit prioritization or proportional resource allocation among processes, resulting in potential performance interference when workloads have differing priorities.

Design implication M3: TSGs and timeslices are the only preemptive hooks. The closed-source firmware hides low-level scheduling details but exposes TSG and timeslice hooks. This unlocks the potential to manipulate TSGs in the global runlist and choose a timeslice per TSG to control kernel preemptions and duty cycles of each application.

[3]An NVIDIA A100-40G GPU has 44MB states (108 SMs, with 256KB registers and 164KB shared memory per SM) and 1.5TB/s HBM bandwidth. Saving old contexts and loading new contexts take $2 \times 44/1.5 \approx 59\mu s$.

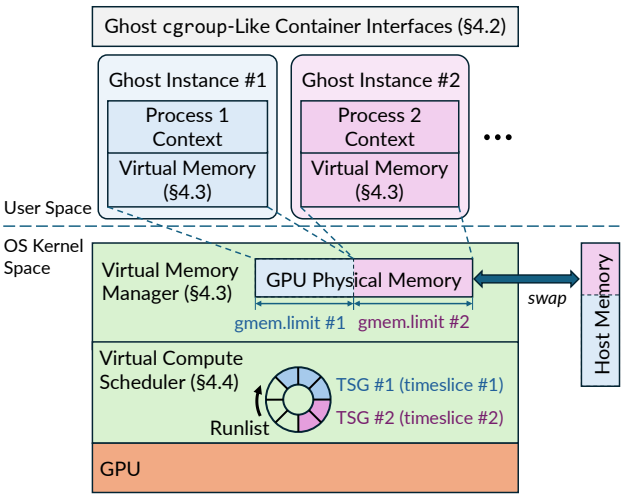


Figure 6: Ghost design overview.

4 Design

Figure 6 presents an overview of Ghost's design.

4.1 The GPU Container Abstraction

Ghost exposes a container-like abstraction and interface surface for GPUs, analogous to CPU and memory control via `cgroup`. Specifically, each process is attached to a per-GPU directory `gcgroup/<pid>/<GPU-id>/`, where the kernel driver exposes interfaces for monitoring and configuring resource limits. As Table 2 summarizes, the `gcgroup` interface defines two categories of controls for memory and compute.

`gmem.limit.high` enforces per-process GPU memory capacity. Once usage exceeds the limit, Ghost transparently evicts pages to host memory; `gmem.limit.low` provides best-effort memory protection: Ghost avoids swapping pages from a process whose `gmem.current` is at or below its `gmem.limit.low`, shielding it from eviction pressure caused by co-located workloads. `gmem.current` and `gmem.swap.current` report the current GPU and host memory usage, respectively. A GPU compute control group resembles a CPU cgroup but manages GPU execution time. `compute.timeslice` specifies the maximum duration a process can occupy the GPU in a scheduling round, while `compute.freeze` allows the operator to instantly freeze or unfreeze a process's kernel execution. `gcgroup.stat` exposes per-process execution statistics, including number of submitted kernels, finished kernels and pending kernels, for monitoring and policy decisions. These parameters can be dynamically adjusted to implement priority-based or weighted-fair-share policies across processes (§4.3). Ghost creates and manages all `gcgroup` directories inside the GPU driver. When a process initializes its CUDA context, Ghost automatically instantiates metadata and exposes the control files. This lightweight, file-based interface enables compatibility with ex-

Interface	Description
`/gmem.limit.high`	Memory usage upper bound (bytes)
`/gmem.limit.low`	Best effort memory protection (bytes)
`/gmem.current`	Current GPU memory usage (bytes)
`/gmem.swap.current`	Current host memory usage (bytes)
`/compute.timeslice`	Max compute time slice (ms) per round
`/compute.freeze`	Freeze / unfreeze kernel execution
`/gcgroup.stat`	Application execution stats

Table 2: gcgroup interfaces for GPU resource management.

isting container infrastructures while allowing kernel-level enforcement of GPU isolation and elasticity.

4.2 Virtual Memory Management

Memory accounting and capacity isolation. Ghost treats all GPU allocations as virtual and manages physical device memory as a shared pool partitioned across containers.

To enforce memory limits and decide when to swap to host memory, Ghost must keep track of how much physical memory is allocated to each container. To achieve this, Ghost transparently redirects applications' memory allocation calls to use UVM interfaces. This ensures that the first access to a page of memory will result in a page fault, giving Ghost a clean way to keep track of each container's memory usage.

When a batch of page faults arrives, Ghost first accounts the new pages to the faulting container and checks its `gmem.limit.high`. If the container is within its limit, Ghost proceeds to bring in the pages. If the container has exceeded its limit or its limit has been reduced, Ghost selects victim pages belonging to the same container, evicts them to host memory, and frees the corresponding device pages until the container returns to its budget. This contrasts with the baseline driver, which evicts pages based only on recency and global pressure and can evict pages across tenants (Figure 7).

To ensure accuracy, Ghost also tracks implicit driver-internal allocations (e.g., page tables, context backing stores) by hooking low-level `ioctl` calls, ensuring `gmem.current` reflects true physical occupancy.

Transparent GPU huge pages. The GPU MMU manages physical pages at 64KB granularity [41] and the driver tracks fine-grained mapping between each 64KB GPU page and its corresponding 16 host OS pages (4KB). For AI workloads with large working set, such fine page granularities can incur high per-page management and PCIe overhead. Ghost tackles this problem with *transparent GPU huge pages*. It opportunistically coalesces contiguous device pages (64KB) and their backing host pages (4KB) into 2MB units and treats these units as the basic granularity for accounting, page-table updates, and I/O. Since we cannot modify GMMU, the GPU page fault is still raise at 64KB boundaries and sent to the driver as a batch. However, Ghost now can opportunistically

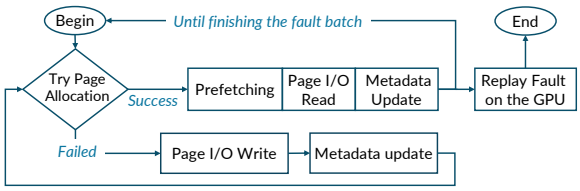


Figure 7: Baseline page-fault handling path in the stock NVIDIA GPU driver. The driver allocates device memory greedily and treats swapping as an exceptional, single-application event.

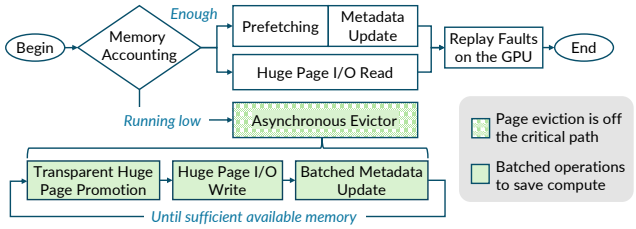


Figure 8: Ghost's page-fault handling path. Ghost enforces per-container limits, uses huge-page-aware batching, and decouples eviction from the critical fault path.

coalesce contiguous faults, allocate a 2MB host huge page, and issue a single 2MB DMA transfer instead of many small 64KB ones. Transparent GPU huge page also simplifies the GPU-host page mapping, and greatly reduces the CPU overhead to update these metadata. For small allocations or the remainder pages that do not align 2MB boundary, Ghost demotes as needed and falls back to the original fault handling path. Promotion and demotion are handled entirely in the driver and are transparent to applications, which continue to see a flat virtual address space.

Overlapped bi-directional swap path. Even with GPU huge pages, swapping on GPUs incurs substantial CPU work to evict pages to make room, bookkeep, update page tables, and issue DMA commands. Even worse, the GPU driver currently executes these operations sequentially (Figure 7), leaving the PCIe, which supports concurrent bi-directional I/O, underutilized in most of the time. Ghost therefore restructures the page-fault path to maximize throughput on both directions and minimize work on the critical path.

As shown in Figure 8, when handling a batch of page faults, Ghost employs asynchrony in both swap-in and eviction paths. Page eviction is moved to a per-container background thread and is automatically triggered when a container's available memory budget running low. Futher, for the swap-in path, Ghost issues (huge) page I/O first, and then continues fault handling to charge gcgroup counters, prefetch pages, and update metadata such as the page table during data transmission. In this way, Ghost can saturate bi-directional PCIe bandwidth and maximize the page swapping throughput.

Victim selection with dual CLOCK lists. Page eviction order is critical to reduce the number of page faults. The

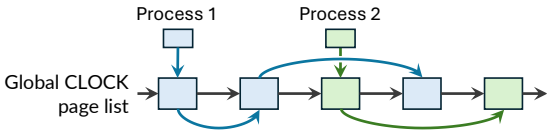


Figure 9: Dual-linked page list. Physical pages form a global CLOCK list, while each container maintains a secondary per-container chain for local eviction decisions.

driver maintains a global CLOCK [11] list and evicts least-recent access pages by default. However, since Ghost evicts pages per container, its eviction order differs from the global eviction order. To balance global memory pressure with per-container isolation, Ghost maintains a dual-linked page list, shown in Figure 9. In addition to the global CLOCK list, each container maintains a secondary chain linking its own pages ordered by access recency. Both lists share the page metadata and have a consistent view of page access recency, but the per-container list can have a different page order than the global list for its own eviction policy.

Ghost walks the chain of a specific container to reclaim its memory, and correspondingly updates the global list along with eviction. but when the device as a whole is under pressure, the evictor can fall back to the global list. This design lets Ghost enforce strict per-container capacity limits while still choosing cold victims, reducing thrashing and preserving throughput under oversubscription.

Application-system collaboration via memory pressure event. Although Ghost heavily optimizes the paging datapath, transferring large working sets across the PCIe bus can still degrade peformance. For many AI workloads, an efficient alternative to demand paging is *rematerialization* [25]—dropping intermediate values (*e.g.*, activation tensors during DNN training or KV cache entries in LLM inference) to free space, and recomputing them when needed.

However, integrating rematerialization into OS-level sharing introduces a fundamental tension: the OS has the global visibility to know *when* memory must be reclaimed, but only the application knows the semantics of *how* to drop and rematerialize specific objects safely. Ghost resolves this through an application-system co-design that establishes a fast, file-descriptor-based notification channel, conceptually similar to Linux's `userfaultfd`. When Ghost's background evictor detects impending memory pressure for a container, it pushes a notification to a registered `usermemfd` that a user-space handler thread can asynchronously `poll()`. This informs the application of the memory reclamation target.

Crucially, to prevent malicious or poorly written applications from stalling the system and breaking hardware isolation, this notification is strictly non-blocking. The driver immediately proceeds to force-evict pages down to the container's capacity limit. Aware applications that quickly process the `usermemfd` event can proactively reduce their memory footprint by dropping rematerializable objects. Conversely, this mechanism strictly preserves Ghost's transparency and isolation: if an application is unmodified, ignores the event, or reacts too slowly, Ghost seamlessly relies on its robust, transparent demand paging mechanisms without compromising the performance of co-located tenants.

4.3 Virtual Compute Scheduler

Ghost virtualizes the GPU compute by *time sharing* all GPU compute resources, including SMs, tensor cores, DMA engines, *etc.*, across running instances. Ghost employs a hybrid scheduler that splits responsibilities between the driver and the hardware (Figure 10). Unlike prior systems like XSched [59], which interpose on every kernel launch and place scheduling logic on the critical path, Ghost offloads fine-grained kernel dispatch entirely to the hardware. This ensures that application streams run at native efficiency without software overhead.

However, moving scheduling decisions off the critical path creates a challenge for isolation: the system must still react instantly when high-priority work arrives. Ghost solves this by uncovering internal driver-firmware RPCs that manipulate the global runlist. These primitives allow the driver to preempt low-priority workloads by shrinking their timeslices and exclude them from the runlist entirely when high-priority containers are active. This collaborative scheduling across driver and is both efficient and flexible.

Non-blocking kernel execution tracking. The scheduler first requires an accurate, low-overhead signal of when a container has "ongoing" kernels (either queued or running). Ghost maintains a count of launched kernels by interposing on the launch API. To track completions and ongoing kernels without blocking, it periodically inserts CUDA events into the stream and probes their status in the background using lightweight memory reads. Subtracting the completion count from the launch count gives the number of ongoing kernels.

Crucially, Ghost uses these events in a novel way solely for *estimation*, not for driving the schedule directly. This distinction allows Ghost to avoid the blocking synchronization required by prior user-space schedulers (§2.3). Instead, Ghost paces event insertion only sparsely (e.g., every 2ms) to align with driver's timeslice granularity and incurs negligible overheads. This approach keeps the tracking logic completely off the execution critical path.

Scheduling primitives. To make scheduling decisions, Ghost extends the driver with a small set of scheduling primitives leveraging two existing driver-GSP RPC interfaces:

- `SetTimeslice`: Programs the maximum execution duration for a TSG. Ghost uses this to force fine-grained preemption. By shrinking the timeslice of a running container, Ghost compels the GSP to preempt running kernels once the timeslice expires.
- `DetachTSG`/`AttachTSG`: Removes or re-inserts a container's

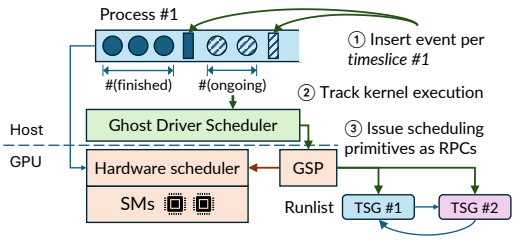


Figure 10: Ghost's driver-hardware collaborative scheduling workflow. The GPU hardware schedules kernels directly for maximum efficiency, while the Ghost driver dynamically adjusts the active runlist to enforce isolation policies.

TSGs into the global hardware runlist. Detaching a TSG effectively suspends the container, ensuring it is completely excluded from consideration by the hardware scheduler.

5 Isolation

A common way to share GPUs today is to pass through an entire device to each OS container [32, 44, 71]. This provides strong isolation but no elasticity: once a container owns a GPU, its unused capacity cannot be reclaimed. Ghost instead virtualizes the GPU in the kernel driver and uses GPU containers as the unit for both compute and memory isolation.

5.1 Isolation Semantics

Container as isolation domain. Each GPU container in Ghost has its own GPU context, page tables, and driver-internal state, and corresponds to one tenant (or tenant group). Kernels or streams within a container are not isolated or tracked separately.

Capacity isolation with oversubscription. On the memory side, Ghost enforces per-container device memory limits (`gmem.limit`) and, when enabled, per-container swap usage. When a container exceeds its configured limit, Ghost evicts that container's own pages to host memory instead of letting it consume unbounded device memory or trigger CUDA OOM errors in other tenants. This gives each container a hard capacity cap plus optional oversubscription, preventing one memory-hungry application from destabilizing others.

Temporal isolation of the full device. On the compute side, Ghost's driver-resident scheduler (§4.3) allocates GPU time across containers using priorities and weights. A key property is that when a container is scheduled, it sees the *entire* GPU: all SMs, copy and codec engines, and full memory bandwidth. Interference arises only at timeslice boundaries, where other containers receive their allotted service. This temporal isolation model differs from hardware partitioning (MIG), where each instance sees only a fraction of the hardware, and is especially beneficial for bursty, latency-sensitive workloads that need short bursts of peak performance.

Fault containment and data safety. For software faults, Ghost provides isolation similar to a dedicated GPU. Each container has its own page tables and driver state; when a process crashes or encounters a GPU error, Ghost tears down only that container, leaving others running. Physical pages are reused across containers, but are reinitialized through the driver's existing allocation path before reuse, so stale data is not exposed to other tenants. Device-wide faults that require a full GPU reset remain global on current hardware and are not masked by Ghost.

5.2 Security Model

Security model. We assume the host OS kernel, vendor GPU kernel driver, Ghost kernel module, and device firmware are trusted. Tenants are treated as mutually untrusted user-space processes that share a physical GPU but possess no kernel privileges. Ghost is designed to prevent these tenants from accessing each other's data or escalating privileges.

Residual risks and comparison to hardware partitioning. Ghost aims to match the isolation level of standard OS processes: tenants cannot read each other's virtual address spaces or GPU memory, and cannot directly crash each other through software faults. However, because Ghost time shares a single monolithic GPU, it cannot eliminate all microarchitectural side channels. Current GPUs do not provide architectural support to fully flush or partition internal structures such as the L2 cache and internal fabrics, so timing and cache-residency channels may remain. In contrast, hardware partitioning mechanisms like MIG spatially split SMs and some cache slices, which can reduce such channels but at the cost of rigid, non-elastic resource assignments. Ghost instead offers MIG-like software and capacity isolation on top of a single instance, while retaining the ability to elastically reallocate compute time and memory across containers.

6 Evaluation

Our evaluation aims to answer the following questions:

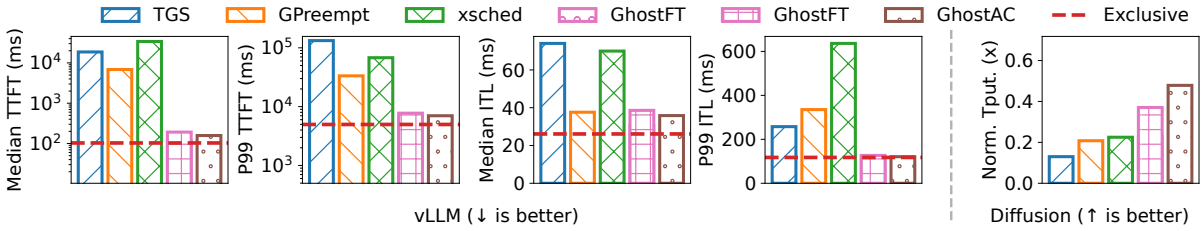
- Does Ghost provide robust performance isolation and adaptively reallocate resources in a way that yields better performance than existing approaches? (§6.3)
- Can Ghost flexibly trade-off latency slackness to best-effort throughput and stay on the Pareto frontier? (§6.4)
- What contributes to Ghost's superior performance? (§6.5)

6.1 Setup

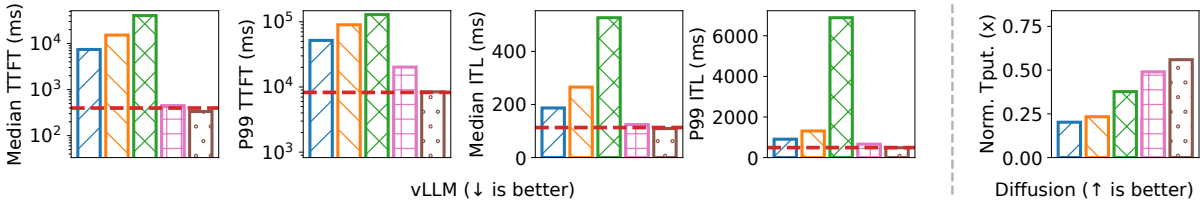
Testbed. We conducted experiments on a GCP instance with one NVIDIA A100-40G GPU and 85GB host memory. The server runs Ubuntu 24.04 (Linux 6.14.0), CUDA 12.9, and NVIDIA open GPU kernel modules 575.57.08.

Baselines. We compare Ghost against state-of-the-art prior work including TGS [71] (transparent memory oversubscription), XSched [59] (user-space scheduling), and GPreempt [16] (kernel-level preemption). We also evaluate industry-standard hardware partitioning using MIG [46].

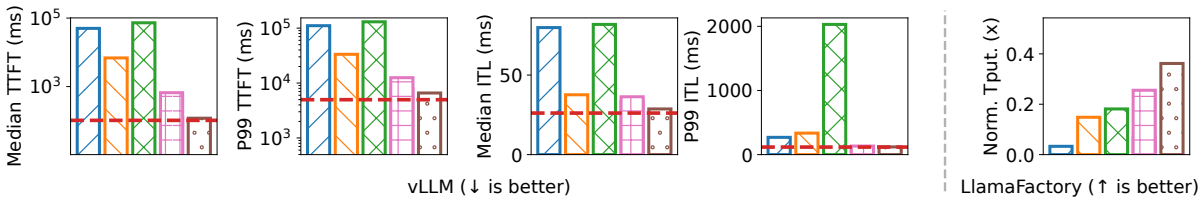
TGS is the only baseline that natively supports memory



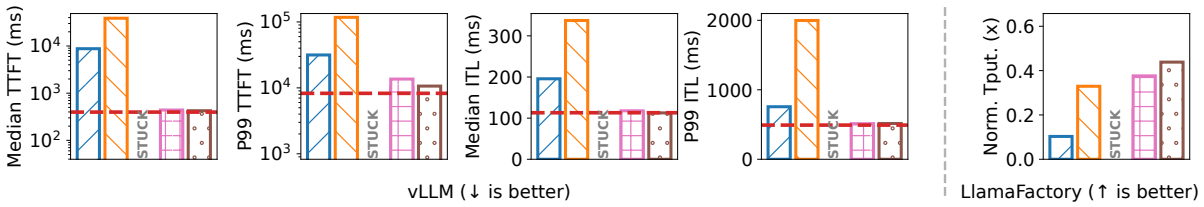
(a) Multimodal AI (Text Pipeline): vLLM (high priority) + Diffusion (low priority).



(b) Multimodal AI (Video Pipeline): vLLM (high priority) + Diffusion (low priority).



(c) Continual online learning (Text Pipeline): vLLM (high priority) + LlamaFactory (low priority).



(d) Continual online learning (Video Pipeline): vLLM (high priority) + LlamaFactory (low priority).

Figure 11: Colocate a high priority (HP) task and a low priority (LP) one under memory pressure. GhostFT uses fully transparent dynamic scheduling, and GhostAC further leverages application cooperation to achieve both near-exclusive HP latency and the highest LP throughput.

overcommitment. For others, we enable memory overcommitment by intercepting their CUDA APIs and routing allocation through UVM, following TGS's approach. Note that TGS builds upon MPS [42] and UVM optimizations, serving as a strictly stronger baseline than standard MPS. We have verified that the TGS consistently achieves the same or higher performance than MPS.

6.2 Applications

We constructed two AI pipelines covering three frameworks, multiple models (LLM, VLM, and diffusion), training and inference, and different priority/SLO requirements. These pipelines reflect emerging trends in multi-modal agentic systems and continual learning workloads. For each pipeline, we evaluate a *text* variant using text-only models and a *video* variant using vision-language and video-generation models.

Multi-modal AI co-locates a latency-critical LLM with a throughput-oriented diffusion model, replicating the resource contention in emerging omni-modal serving frameworks [63, 64]. The text variant pairs Llama-3.1 8B (vLLM [33], high priority) with Stable Diffusion 3.5 Medium (Diffusers [2], low priority); the video variant uses Qwen2.5-VL-3B [8] for video-to-text inference and WAN2.1-Small [66] for video generation. We use BurstGPT [68] traces for LLM/VLM and VidProM [67] prompts for diffusion.

Continual online learning serves an LLM for both inference and LoRA finetuning with periodical weight updates [26, 75]. The text variant pairs Llama-3.2-3B on vLLM [33] (high priority) with Llama-Factory [83] for finetuning (best-effort); the video variant uses Qwen2.5-VL-3B [8] for both serving and training. Following prior studies [79], we use BurstGPT [68] for inference, Alpaca [61] for

text finetuning, and MMVU [82] for video finetuning.

6.3 End-to-End Performance

6.3.1 Performance Isolation. We run both pipelines in a memory-constrained configuration to test whether Ghost can preserve high-priority performance while maintaining useful low-priority throughput. We evaluate two policies built on Ghost's `gcgroup` interface: GhostFT adds a fully transparent (FT) dynamic scheduler that adjusts the LP task's memory limit based on HP workload intensity; GhostAC further incorporates application cooperation (AC), allowing the LP task to proactively yield resources. Results are shown in Figure 11. When co-locating vLLM (HP) and Diffusion (LP) in the text pipeline (Figure 11b), GhostAC achieves both near-exclusive vLLM latency and the highest diffusion throughput among all systems. Compared to the best-performing baseline, GhostAC reduces vLLM median and P99 TTFT by 43× and 4.8×, and P99 ITL by 2.8×, while keeping P99 ITL within 3% of the exclusive baseline. GhostFT achieves comparable HP isolation while trading modest latency for improved LP throughput. The video pipeline shows a similar trend: GhostAC matches the exclusive vLLM latency while delivering 1.5× higher diffusion throughput than the best baseline.

All baselines experience high vLLM latency due to the lack of memory isolation. Similar to memory contention shown in Figure 2, the colocated LP task can swap out vLLM memory during execution, leading to latency spikes. They also suffer from low LP throughput due to inefficient paging with the default GPU driver.

Co-locating vLLM and LlamaFactory (Figure 11d) shows a similar trend. GhostAC achieves vLLM latency within 16% of exclusive while delivering the highest LP throughput, outperforming the best baseline by 2×. GhostFT trades HP latency for further LP throughput gains. In the video pipeline, XSched becomes stuck entirely, while GhostAC maintains near-exclusive vLLM latency and the best LP throughput.

Using the same `gcgroup`-like interface, we also implement a dynamic policy, GhostDYN, which dynamically adjusts and relaxes the memory limit for the LP task to increase utilization, when vLLM has a low load that cannot fully saturate its reservation. As Figure 11b and Figure 11d show, GhostDYN slightly increases vLLM latency relative to Ghost that enforces strict isolation, but still outperforms all baselines. In return, GhostDYN improves LP throughput by an average of 2×, indicating that modest slack in HP latency can be traded for substantial LP throughput gains.

6.3.2 GPU Compute Efficiency. We next isolate the impact of Ghost's compute virtualization by running the same co-location experiments on an A100-80G GPU, so that memory contention is effectively removed. We keep the other hardware and software setups the same to ensure this configuration exercises only compute-related mechanisms. Results are shown in Figure 12.

In this case, GhostFT achieves the lowest latency for vLLM (the high-priority task) across both pipelines. Specifically, GhostFT significantly outperforms MIG and achieves up to 480×, 184×, 2.75×, and 2.0× lower latencies for median and P99 TTFT and ITL, respectively, across both pipelines. MIG performs worst in this case due to the NVIDIA hardware constraints that it can allocate up to 50% of GPU compute and memory to vLLM, which is insufficient for vLLM to handle its peak load. This highlights that Ghost's driver-hardware collaborative scheduling provides much more flexible resource limit configurations than MIG and stronger isolation than the other baselines. For LP task throughput, GhostFT performs slightly worse than MIG but outperforms all other baselines. Compared to MIG, GhostFT achieves 88% Diffusion throughput for the multimodal AI workload and 88% LlamaFactory throughput for continual learning workload. MIG achieves the best LP throughput because it allocates the rest 50% GPU resources to the LP task. TGS achieves low HP latency but also much lower LP throughput because it conservatively stops scheduling LP kernels to provide compute isolation to vLLM. GPreempt and XSched, in contrast, aim for high overall throughput but fail to provide strong enough isolation to keep vLLM latency low.

Overall, GhostFT achieves an ideal balance between HP and LP tasks by keeping consistently low HP latency while significantly improving LP throughput.

6.4 Dynamic Resource Coordination

In this experiment, we evaluate Ghost's elasticity by dynamically adjusting resource limit and allocation between applications and investigate whether can constantly achieve Pareto frontier compared to baseline systems.

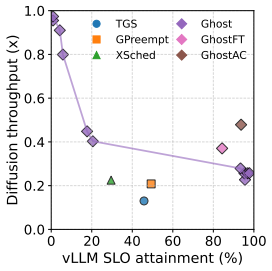
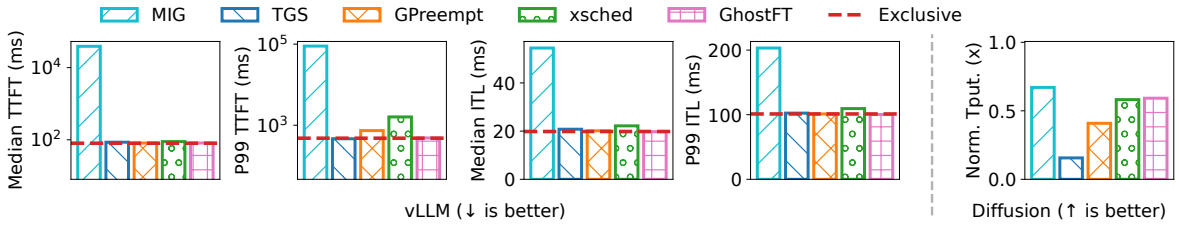


Figure 14: vLLM SLO attainment vs. diffusion throughput.

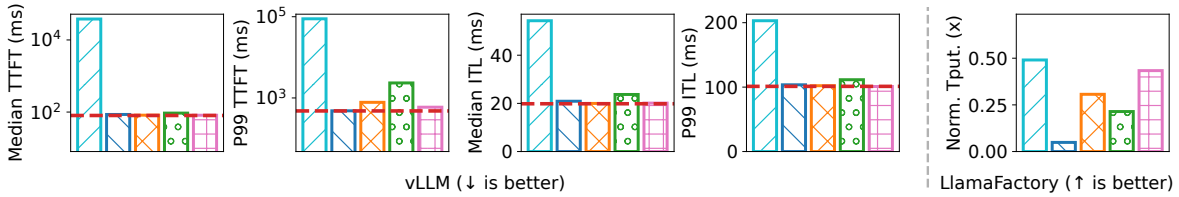
Figure 14 plots SLO attainment against normalized diffusion throughput when co-locating vLLM (HP) and Diffusion (LP). We define the SLO target as vLLM's P99 latency when running alone, and normalize Diffusion throughput to its ideal throughput on a dedicated GPU.

None of the baselines achieve more than 80% SLO attainment or 0.35× normalized throughput. This aligns with earlier results: they neither enforce memory isolation nor implement efficient scheduling and paging under contention.

Ghost, in contrast, achieves 100% SLO attainment while matching the best LP throughput among baselines. More importantly, Ghost can adjust resource allocation according to HP latency requirements. By varying the compute duty



(a) Multimodal AI: vLLM (high priority) + Diffusion (low priority)



(b) Continual online learning: vLLM (high priority) + LlamaFactory (low priority)

Figure 12: Performance variation when co-running vLLM with (a) Diffusion and (b) LlamaFactory when memory is sufficient.

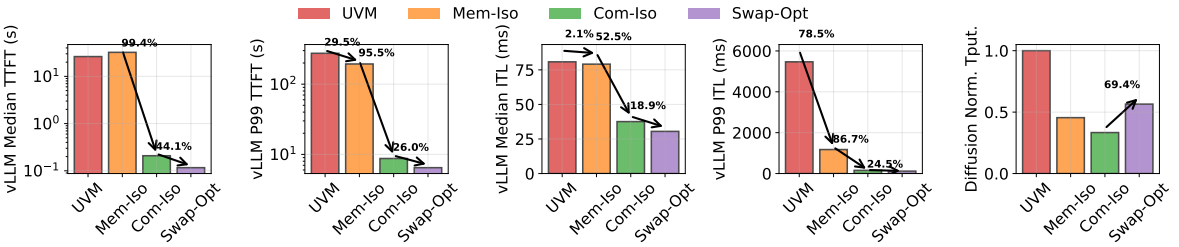


Figure 13: All three of Ghost's contributions work in tandem to improve latency and throughput.

cycle of the LP task, Ghost trades off vLLM latency against diffusion throughput and eventually reaches full LP throughput. Across this range, Ghost remains on the Pareto frontier, indicating that no other system dominates it on both metrics.

GhostFT and GhostAC push this frontier further outward by dynamically reallocating GPU memory based on the current workload. GhostAC maintains over 93% SLO attainment while achieving 0.48× normalized LP throughput—a level that static Ghost reaches only when its SLO attainment drops below 20%. This demonstrates the effectiveness of coordinated adjustment of both compute and memory limits, especially when combined with application cooperation, in improving utilization.

6.5 Performance Analysis

Ablation Study. We incrementally enable Ghost's mechanisms while co-locating vLLM and Diffusion under the same memory-constrained setting as in Figure 11. Figure 13 summarizes the results.

Adding memory isolation alone protects vLLM from heavy paging, substantially reducing vLLM tail latency by 29.5% for P99 TTFT and 78.5% for P99 ITL. Enabling compute isolation prioritizes vLLM's kernels, further reducing median and P99 TTFT by 99.4% and 95.5%, and median and P99 ITL by 52.5% and 86.7%, respectively. Finally, enabling Ghost's optimized paging increases Diffusion throughput by 69.4%, which previously suffered from slow swapping. Interestingly, efficient paging also reduces vLLM latency. Our inspection reveals that the GPU cannot promptly switch process contexts until outstanding page faults are resolved, so a swap-heavy process can delay other processes even when the scheduler nominally assigns them proportional compute time. By reducing page-fault and swap time, Ghost removes this hidden source of interference.

Microbenchmarks. We further quantify the impact of Ghost's swapping optimizations on low-priority tasks under memory contention. Figure 15 shows the time breakdown of diffusion when its available GPU memory is capped to 40% of its nominal requirement. With Ghost, memory swapping overhead is comparable to compute time, resulting in a manageable 2× throughput reduction. In contrast, the unoptimized baseline suffers a severe drop (>15× in Figure 3) due to synchronous blocking and small-packet PCIe congestion. Figure 16 plots diffusion throughput versus its GPU memory usage. Notably, Ghost consistently outperforms the unoptimized baseline across all configurations, and can offload 10% of its working set with minor throughput drop.

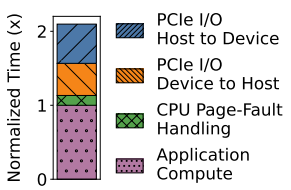


Figure 15: Diffusion time breakdown under memory pressure.

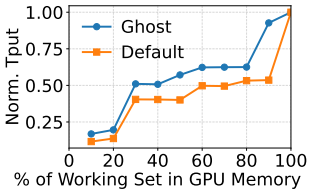


Figure 16: Normalized Diffusion throughput vs. memory usage.

7 Related Work

Software GPU partitioning. In addition to hardware partitioning solutions discussed in §2.2, systems such as HAMi [21], run:ai [58], and fractional GPU [3] intercept CUDA APIs to allow applications to use a subset of GPU resources. These approaches statically divide GPU capacity, lack elasticity, and assume all workloads fit in device memory. In contrast, Ghost virtualizes GPUs inside the kernel driver to support dynamic reallocation and strong isolation.

GPU kernel scheduling. Many GPU kernel schedulers have been proposed for efficient compute sharing. Systems such as Paella [37], Orion [60], Reef [22], and others [17, 20, 53, 56, 72, 76], coordinate kernel launches across processes to improve utilization. Bless [80], Tally [81], and LithOS [13], anothers [10, 38, 52], introduce finer-grained SM partitioning. These efforts focus primarily on compute sharing and do not address memory management or fault isolation.

GPU virtual memory and oversubscription. Recent vendor APIs [6, 24, 43] expose low-level virtual memory interfaces that allow user programs to manually manage physical page mappings, but provide no protection against mismanagement. Application-level systems such as PipeSwitch [9], SwapAdvisor [23], Capuchin [55], and others [35, 36, 74], support memory offloading but rely on user-space control, provide weak isolation, and target specific workloads. Other GPU swapping systems [12, 29–31] improve prefetching or exploit faster interconnects, but require application modifications and still lack isolation. To our knowledge, Ghost is the first system to provide both strong memory isolation and safe oversubscription for general GPU workloads.

8 Conclusion

Ghost enables strongly isolated and elastic GPU sharing by virtualizing GPUs inside the OS kernel. It combines kernel-level control of compute and memory with lightweight scheduling and memory management to ensure safe, efficient sharing without hardware modification. Evaluations across diverse AI workloads show that Ghost significantly improves GPU utilization while preserving predictable performance.

References

[1] How we built our multi-agent research system — anthropic.com. https://www.anthropic.com/engineering/built-multi-agent-research-system. [Accessed 14-07-2025].

[2] Hugging Face Diffusers. https://huggingface.co/docs/diffusers/en/index. [Accessed 04-10-2025].

[3] N. Agarwal. Implementing Fractional GPUs in Kubernetes with Aliyun Scheduler. https://huggingface.co/blog/NileshInfer/implementing-fractional-gpus-in-kubernetes, 2024. Accessed: August, 2025.

[4] AMD. AMD Vega Unified Memory. https://rocm.docs.amd.com/projects/HIP/en/docs-6.2.0/how-to/unified_memory.html. [Accessed 08-11-2025].

[5] AMD. HIP Runtime API Reference: Managed Memory. https://rocm.docs.amd.com/projects/HIP/en/docs-6.0.0/doxygen/html/group___memory_m.html. [Accessed 01-11-2025].

[6] AMD. HIP Runtime API Reference: Virtual Memory Management. https://rocm.docs.amd.com/projects/HIP/en/latest/doxygen/html/group___virtual.html. [Accessed 01-11-2025].

[7] AMD. ROCm Documentation. https://rocm.docs.amd.com/en/latest/. [Accessed 01-11-2025].

[8] S. Bai, K. Chen, X. Liu, et al. Qwen2.5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025.

[9] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514. USENIX Association, Nov. 2020.

[10] J. Bakita and J. H. Anderson. Hardware Compute Partitioning on NVIDIA GPUs. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 54–66, 2023.

[11] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[12] S. Choi, T. Kim, J. Jeong, R. Ausavarungnirun, M. Jeon, Y. Kwon, and J. Ahn. Memory harvesting in Multi-GPU systems with hierarchical unified virtual memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 625–638, Carlsbad, CA, July 2022. USENIX Association.

[13] P. H. Coppock, B. Zhang, E. H. Solomon, V. Kypriotis, L. Yang, B. Sharma, D. Schatzberg, T. C. Mowry, and D. Skarlatos. Lithos: An operating system for efficient machine learning on gpus. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 1–17, New York, NY, USA, 2025. Association for Computing Machinery.

[14] DeepSeek-AI et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[15] Y. Du, S. Li, A. Torralba, J. B. Tenenbaum, and I. Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *Forty-first International Conference on Machine Learning*, 2024.

[16] R. Fan, T. Ren, M. Xie, S. Gao, J. Shu, and Y. Lu. Gpreempt: Gpu preemptive scheduling made general and efficient. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '25, USA, 2025. USENIX Association.

[17] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In P. D'Ambra, M. Guarracino, and D. Talia, editors, *Euro-Par 2010 - Parallel Processing*, pages 379–391, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[18] J. Gu, S. Song, Y. Li, and H. Luo. Gaiagpu: Sharing gpus in container clouds. In *2018 IEEE ISPA/IUCC/BDCloud/SocialCom/SustainCom*, pages 469–476, 2018.

[19] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang. Large language model based multi-agents: A survey of progress and challenges, 2024.

[20] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-Level Virtualization for High Performance Computing*, HPCVirt '09, page 17–24, New York,

NY, USA, 2009. Association for Computing Machinery.

[21] HAMi. Project-HAMi/HAMi: Heterogeneous AI Computing Virtualization Middleware. https://github.com/Project-HAMi/HAMi. [Accessed 31-10-2025].

[22] M. Han, H. Zhang, R. Chen, and H. Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[23] C.-C. Huang, G. Jin, and J. Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1341–1355, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Intel. Level Zero Specification documentation: Reserving virtual address space. https://oneapi-src.github.io/level-zero-spec/level-zero/latest/core/PROG.html#reserved-device-allocations. [Accessed 01-11-2025].

[25] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, K. Keutzer, I. Stoica, and J. E. Gonzalez. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *MLSys*. MLSys, 2020.

[26] D. Jayasuriya, S. Tayebati, D. Ettori, R. Krishnan, and A. R. Trivedi. Sparc: Subspace-aware prompt adaptation for robust continual learning in llms, 2025.

[27] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.

[28] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong, Y. Jia, S. He, H. Chen, Z. Bai, Q. Hou, S. Yan, D. Zhou, Y. Sheng, Z. Jiang, H. Xu, H. Wei, Z. Zhang, P. Nie, L. Zou, S. Zhao, L. Xiang, Z. Liu, Z. Li, X. Jia, J. Ye, X. Jin, and X. Liu. Megascale: scaling large language model training to more than 10,000 gpus. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI'24, USA, 2024. USENIX Association.

[29] J. Jung, J. Kim, and J. Lee. 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*, ASPLOS 2023, page 207–221, New York, NY, USA, 2023. Association for Computing Machinery.

[30] W. Kang, J. Lee, Y. Lee, S. Oh, K. Lee, and H. S. Chwa. Rt-swap: Addressing gpu memory bottlenecks for real-time multi-dnn inference. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 373–385, 2024.

[31] J. Kehne, J. Metter, and F. Bellosa. Gpuswap: Enabling oversubscription of gpu memory through transparent swapping. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, page 65–77, New York, NY, USA, 2015. Association for Computing Machinery.

[32] Kubernetes. Dynamic Resource Allocation. https://kubernetes.io/docs/concepts/scheduling-eviction/dynamic-resource-allocation/, 2025. Accessed: August, 2025.

[33] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[34] Ling-Team and . others. Every Activation Boosted: Scaling General Reasoner to 1 Trillion Open Language Foundation. *arXiv preprint arXiv:2510.22115*, 2025.

[35] LMCache Team. LMCache. https://lmcache.ai/. [Accessed 01-11-2025].

[36] Moonshot AI. Mooncake KV Cache Transfer Engine. https://github.com/kvcache-ai/Mooncake/, 2024.

[37] K. K. W. Ng, H. M. Demoulin, and V. Liu. Paella: Low-latency model serving with software-defined gpu scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 595–610, New York, NY, USA, 2023. Association for Computing Machinery.

[38] NVIDIA. CUDA green contexts. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__GREEN__CONTEXTS.html. [Accessed 01-11-2025].

[39] NVIDIA. CUDA Managed Memory. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html. [Accessed 01-11-2025].

[40] NVIDIA. NVIDIA Pascal Unified Memory Improvement. https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#unified-memory-improvements. [Accessed 08-11-2025].

[41] NVIDIA. Pascal mmu format changes. https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf. [Accessed 14-07-2025].

[42] NVIDIA. NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html, 2024. Accessed: August, 2025.

[43] NVIDIA. Cuda toolkit documentation—virtual memory management. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__VA.html, 2025. Accessed: August, 2025.

[44] NVIDIA. NVIDIA Kubernetes Device Plugin. https://catalog.ngc.nvidia.com/orgs/nvidia/containers/k8s-device-plugin, 2025. Accessed: August, 2025.

[45] NVIDIA. NVIDIA Linux open GPU kernel module source. https://github.com/NVIDIA/open-gpu-kernel-modules, 2025. Accessed: August, 2025.

[46] NVIDIA. NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2025. Accessed: August, 2025.

[47] NVIDIA. Nvlink & nvswitch for advanced multi-gpu communication — nvidia.com. https://www.nvidia.com/en-us/data-center/nvlink/, 2025. Accessed: August, 2025.

[48] NVIDIA. Unlock Next Level Performance with Virtual GPUs. https://www.nvidia.com/en-us/data-center/virtual-solutions/, 2025. Accessed: August, 2025.

[49] NVIDIA. Nvidia dgx b200 specifications. https://www.nvidia.com/en-us/data-center/dgx-b200/, 2026.

[50] OpenAI. Gpt-4o system card, 2024.

[51] OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025.

[52] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 407–418, New York, NY, USA, 2013. Association for Computing Machinery.

[53] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 593–606, New York, NY, USA, 2015. Association for Computing Machinery.

[54] J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, UIST '23, New York, NY, USA, 2023. Association for Computing Machinery.

[55] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 891–905, New York, NY, USA, 2020. Association for Computing Machinery.

[56] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, page 217–228,

New York, NY, USA, 2011. Association for Computing Machinery.
[57] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-resolution image synthesis with latent diffusion models, 2021.
[58] Run:ai. Quickstart: Launch Workloads with GPU Fractions. https://docs.run.ai/v2.17/Researcher/Walkthroughs/walkthrough-fractions/, 2023. Accessed: August, 2025.
[59] W. Shen, M. Han, J. Liu, R. Chen, and H. Chen. Xsched: preemptive scheduling for diverse xpus. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.
[60] F. Strati, X. Ma, and A. Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1075–1092, 2024.
[61] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.
[62] K. Team. Kimi k2: Open agentic intelligence, 2025.
[63] S.-O. Team. Sglang-omni. https://docs.sglang.io/sglang-omni/, 2026.
[64] vLLM Omni Team. vllm-omni. https://docs.vllm.ai/projects/vllm-omni/en/latest/, 2026.
[65] B. Wan, M. Han, Y. Sheng, Y. Peng, H. Lin, M. Zhang, Z. Lai, M. Yu, J. Zhang, Z. Song, X. Liu, and C. Wu. Bytecheckpoint: a unified checkpointing system for large foundation model development. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '25, USA, 2025. USENIX Association.
[66] A. Wang, B. Ai, B. Wen, C. Mao, et al. Wan: Open and advanced large-scale video generative models. *arXiv preprint arXiv:2503.20314*, 2025.
[67] W. Wang and Y. Yang. Vidprom: A million-scale real prompt-gallery dataset for text-to-video diffusion models, 2024.
[68] Y. Wang, Y. Chen, Z. Li, Z. Tang, R. Guo, X. Wang, Q. Wang, A. C. Zhou, and X. Chu. Towards efficient and reliable llm serving: A real-world workload study, 2024.
[69] H. Wei, Y. Sun, and Y. Li. Deepseek-ocr: Contexts optical compression. *arXiv preprint arXiv:2510.18234*, 2025.
[70] Q. Weng, L. Yang, Y. Yu, W. Wang, X. Tang, G. Yang, and L. Zhang. Beware of fragmentation: Scheduling GPU-Sharing workloads with fragmentation gradient descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 995–1008, Boston, MA, July 2023. USENIX Association.
[71] B. Wu, Z. Zhang, Z. Bai, X. Liu, and X. Jin. Transparent GPU sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, Boston, MA, Apr. 2023. USENIX Association.
[72] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, Nov. 2020.
[73] J. Xu, Z. Guo, H. Hu, Y. Chu, X. Wang, J. He, Y. Wang, X. Shi, T. He, X. Zhu, Y. Lv, Y. Wang, D. Guo, H. Wang, L. Ma, P. Zhang, X. Zhang, H. Hao, Z. Guo, B. Yang, B. Zhang, Z. Ma, X. Wei, S. Bai, K. Chen, X. Liu, P. Wang, M. Yang, D. Liu, X. Ren, B. Zheng, R. Men, F. Zhou, B. Yu, J. Yang, L. Yu, J. Zhou, and J. Lin. Qwen3-omni technical report, 2025.
[74] Y. Xu, Z. Mao, X. Mo, S. Liu, and I. Stoica. Pie: Pooling cpu memory for llm inference, 2024.
[75] M. Yang, F. Yang, Y. Guo, S. Xu, T. Zhou, Y. Chen, S. Shao, J. Liu, and Y. Gao. Pcl: Prompt-based continual learning for user modeling in recommender systems. In *Companion Proceedings of the ACM on Web Conference 2025*, WWW '25, page 1475–1479. ACM, May 2025.
[76] P. Yu and M. Chowdhury. Salus: Fine-grained gpu sharing primitives for deep learning applications, 2019.
[77] S. Yu, J. Xing, Y. Qiao, M. Ma, Y. Li, Y. Wang, S. Yang, Z. Xie, S. Cao, K. Bao, I. Stoica, H. Xu, and Y. Sheng. Prism: Unleashing gpu sharing for cost-efficient multi-llm serving, 2025.
[78] M. Zaharia, O. Khattab, L. Chen, J. Q. Davis, H. Miller, C. Potts, J. Zou, M. Carbin, J. Frankle, N. Rao, and A. Ghodsi. The shift from models to compound ai systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/, 2024.
[79] D. Zhang, H. Wang, Y. Liu, X. Wei, Y. Shan, R. Chen, and H. Chen. Blitzscale: fast and live large model autoscaling with o(1) host caching. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.
[80] S. Zhang, Q. Chen, W. Cui, H. Zhao, C. Xue, Z. Zheng, W. Lin, and M. Guo. Improving gpu sharing performance through adaptive bubbleless spatial-temporal sharing. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 573–588, New York, NY, USA, 2025. Association for Computing Machinery.
[81] W. Zhao, A. Jayarajan, and G. Pekhimenko. Tally: Non-intrusive performance isolation for concurrent deep learning workloads. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 1052–1068, New York, NY, USA, 2025. Association for Computing Machinery.
[82] Y. Zhao, L. Xie, H. Zhang, et al. Mmvu: Measuring expert-level multi-discipline video understanding. *arXiv preprint arXiv:2501.12380*, 2025.
[83] Y. Zheng, R. Zhang, J. Zhang, Y. Ye, Z. Luo, Z. Feng, and Y. Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics.

A Appendix

A.1 Implementation

We implemented Ghost by extending NVIDIA open-source kernel modules with 5K lines of C code. Ghost currently supports NVIDIA GPUs later than the Pascal architecture. We plan to support AMD GPUs in the future, which also have open-sourced drivers [7] that share a similar structure.

Ghost is easy to use and it integrates with standard Linux interfaces in two ways. First, we provide optional kernel patches that extend the native Linux `cgroup` subsystem, allowing GPU resources to be managed seamlessly alongside CPU and memory. Second, to support deployments where modifying the host kernel is undesirable, Ghost implements a parallel interface over Linux `debugfs` that mimics the standard `cgroup` file-system structure.

Ghost employs a thin user-space shim (similar to prior work [59, 71]) to hook CUDA calls. It redirects memory allocations (*e.g.*, `cuMemAlloc`) to Ghost's modified UVM backend (§4.2) to enable transparent paging, while instrumenting kernel launches (*e.g.*, `cuLaunchKernel`) to track execution status. Inside the driver, Ghost extends the existing UVM infrastructure, reusing native page tables while injecting custom logic for per-container accounting, isolation, and optimized bi-directional swapping.

A.2 Scheduling Policies

Ghost implements simple container-level policies that mirror common CPU abstractions. Users can also customize policies

with exposed `gcgroup` interfaces.

Priority-based scheduling. Each container is assigned a priority. Ghost ensures that higher-priority containers run before lower-priority ones by attaching their TSGs to the runlist and shrinking the timeslice or detaching the TSG of lower-priority containers when contention arises. High-priority containers thus can run with guaranteed performance, while lower-priority work uses whatever capacity remains.

Weighted-fair scheduling. For throughput sharing, Ghost supports weighted fairness, corresponding to fractional GPUs. Each container has a weight that encodes its target share. Ghost uses a lightweight weighted round-robin scheme: containers accumulate credit in proportion to their weight, and any container with positive credit is scheduled for a bounded timeslice; the time used is then deducted from its credit. As long as some container has work, the GPU stays busy, and over time each container receives service roughly proportional to its weight.

A.3 Limitations

First, Ghost relies the user or operator to appropriately set and adjust the parameters on each container (e.g., memory limits and compute priorities). Our evaluation focuses on preserving the performance of a single high-priority application in the face of contention. While Ghost can also be applied in other settings, it does not resolve the fundamental policy question of which application to prioritize when resources are scarce. Second, the use of virtual memory and on-demand physical page allocation will incur overhead upon the first access to allocated virtual addresses. This happens most during initialization when allocation is frequent and has no effect during kernel execution. For most AI workloads that do not allocate memory frequently, this overhead should be negligible. Finally, Ghost does not offer spatial SM sharing to enforce strong isolation, but users are free to do so atop Ghost with tools such as CUDA Green Context [38] and LithOS [13].