GVM: OS-Level GPU Virtualization for Mixing Interactive and Batch Workloads

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

[1]*UC Berkeley* [2]*UCLA* *

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 compromise. They either partition hardware for strong isolation but sacrifice elasticity, or employ application-level resource sharing for flexibility but fail to guarantee memory isolation or fault containment. We present GVM, an OS-level GPU virtualization layer integrated directly into the open-source GPU driver. GVM achieves hardware-like performance isolation while preserving the flexibility of software-based sharing through three mechanisms: (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, GVM improves throughput by up to 2× and reduces latency by up to 59× across two representative AI frameworks. GVM is publicly available at https://github.com/ovg-project/GVM.

1 Introduction

GPUs are costly but often underutilized in production. Recent hardware trends show rapid growth in capacity; for example, the NVIDIA B200 features 192GB of HBM and costs approximately $500k. Yet, modern AI workloads are increasingly diverse. Many specialized models are too small or lack sufficient traffic to saturate such high-capacity devices. As a result, major datacenters report that GPU utilization typically remains below 30% for compute and below 60% for memory [13, 67].

Resource virtualization and GPU sharing among multiple applications is a natural way to improve utilization [13, 24, 38, 43, 47, 49, 60, 76]. This is particularly relevant for modern compound AI systems [50, 70, 74] and multi-agent frameworks [1, 15, 21, 54]. These systems combine multiple distinct components, For instance, GPT-oss-120B [51] consumes ~65GB HBM, whereas smaller models like GPT-oss-20B [51] or Qwen3-Omni talker [70] require only 10–13GB. A single high-end GPU like H100 or B200 can theoretically colocate these models to achieve much higher density.

Effective sharing requires *elasticity*: the ability to dynamically reassign underutilized resources to other applications. But achieving elasticity without sacrificing *isolation* remains a fundamental OS challenge [10, 16, 18]. Elasticity often introduces *noisy neighbor* effects and failure propagation. When multiple applications run under memory pressure, their total demand can temporarily exceed GPU capacity, and a single process allocating excessive memory can trigger CUDA out-of-memory errors that crash co-located applications. Even with systems that support memory oversubscription [5, 40], contention for device 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.

However, existing GPU-sharing mechanisms force a trade-off between these two properties. *Hardware-level partitioning mechanisms* provided by GPU vendors, including NVIDIA MIG [47] and vGPU [49], offer strong isolation by statically dividing GPU compute and memory. However, modifying these partitions typically requires a full GPU reset and termination of running applications, rendering them unsuitable for dynamic workloads. Conversely, *application-level solutions*, such as NVIDIA MPS [43] and recent research systems [13, 20, 24, 38, 60, 76], enable elastic compute scheduling by intercepting or modifying an application's kernel launching calls to reorder kernels. However, since they run on the host, they lack visibility into fresh kernel execution status and

*Equal contributions, listed in alphabetical order. †Corresponding author.

hardware isolation mechanisms, and hence cannot schedule kernels as efficiently as the GPU hardware scheduler, nor provide memory isolation.

This dichotomy between elasticity and isolation stems from a gap between GPU hardware and applications. On one hand, modern GPU hardware has evolved into managed devices with rich resource control mechanisms, such as preemption handlers, scheduling hooks, and page tables; on the other hand, GPUs expose only narrow, high-level interfaces to the host OS and applications, hiding the low-level GPU management logic in their proprietary driver code. As a result, sharing systems face a binary choice: 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.

The recent release of open-source GPU drivers [7, 46] provides a new opportunity. For the first time, the host OS can access low-level GPU mechanisms for virtual memory management and scheduling. However, these raw interfaces are not immediately usable, as safely manipulating interrupt handlers and page tables requires privileged execution and careful coordination, and the exposed interfaces are too low-level to function as a practical sharing abstraction for applications.

In this paper, we explore whether implementing GPU sharing in the GPU driver running in the OS kernel, instead of at the application or in the hardware, can achieve a fundamentally more favorable trade-off between elasticity and isolation. Our core insight is that we can decouple resource isolation from hardware partitioning with newly available low-level interfaces, essentially turning the open driver into a "resource hypervisor" to flexibly virtualize and multiplex the GPU.

Building on this insight, we introduce GVM, an OS-level GPU virtualization layer integrated into the open-source GPU kernel driver [46]. GVM provides applications a *GPU container* abstraction with `cgroup`-like interfaces to specify compute and memory quotas. It enforces these limits using newly exposed hardware mechanisms (*e.g.*, preemption, scheduling hooks, and page fault handlers) while dynamically redistributing unused resources. Unlike application-level approaches, GVM runs with OS kernel privileges, which enables precise memory accounting, enforced memory residency, and fault containment. In contrast to hardware partitioning, GVM requires no firmware changes or device resets. Instead, it achieves elasticity by continuously monitoring GPU state and adapting resource allocations within the driver, without requiring GPU resets or interrupting running applications. Crucially, GVM is fully transparent to applications, requiring no source code modifications or recompilation.

Challenges. Designing and implementing GVM requires overcoming two challenges. These challenges are rooted in the limited interface between the driver (on the CPU) and the GPU device, and the fact that while the GPU driver is programmable, the GPU firmware remains proprietary.

The first challenge is to enable the GPU driver to have page-level control of GPU memory, so it can achieve elasticity and isolation. This is challenging because the GPU driver cannot normally access the page table or observe page faults; instead, only the proprietary GPU firmware has access to this. Our solution has two steps. First, we observe that the GPU driver can see page faults for memory allocated with Unified Virtual Memory (UVM) [5, 40] and update their page mappings. Therefore, we intercept memory allocation calls and redirect them to use UVM to obtain page fault observability. Because all allocations now use UVM, we gain a clean hook for tracking memory usage and enforcing memory limits (by paging out). Second, we speed up UVM memory swapping via a series of optimizations, including asynchronous page fault handling to overlap page in and out, huge pages to batch DMA operations, and a redesigned LRU list to reduce CPU overhead. GVM is the first system to realize these optimizations in the GPU driver, achieving a 2× end-to-end speedup under memory contention (§4.3).

The second challenge is to design an efficient and expressive scheduling framework. Implementing a classic OS scheduler for GPU kernels in the driver is inefficient due to the high frequency of kernels (often complete in <100µs), which would trigger excessive remote procedure calls (RPCs) to the GPU across the PCIe. Customizing the GPU firmware is an attractive alternative, but the firmware is proprietary. Instead, GVM makes 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. On the other hand, we identify internal firmware RPC interfaces, which are previously unknown, to enforce high-level control such as determining the timeslice and triggering preemption.

This allows our scheduling framework to inject scheduling decisions only when necessary, which minimizes PCIe overhead and outperforms state-of-the-art schedulers (§4.4).

We implemented GVM for NVIDIA GPUs and evaluated it on an A100 GPU using two representative AI workloads. Compared to the state of the art, GVM effectively isolates colocated applications, i.e., reducing latency-sensitive applications' latency by up to 59× while improving throughput-oriented applications' throughput by up to 2×.

2 Background and Motivation

2.1 GPU Utilization Challenges

Various cloud providers, including Microsoft [28], Meta [13], and Alibaba [67], report low GPU utilization. As shown in Figure 1, nearly half of GPU memory remains stranded in an Alibaba production cluster. GPU compute utilization also fluctuates and stays below 25%[1] for most of the time.

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

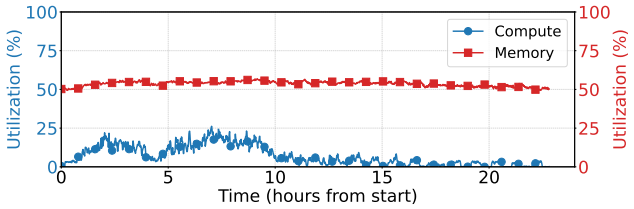


Figure 1: Alibaba's production GPU cluster shows low utilization [67], with compute <30% and memory <60% most of the time.

Compute underutilization. The temporal characteristics of real-world workloads make it challenging to maintain high average compute utilization. For inference workloads, utilization is dictated by service-level demand, which is highly variable. Real-world traces indicate that request arrival rates can fluctuate by more than 3× within tens of seconds [65]. Consequently, the GPU remains completely idle during the frequent gaps between requests. Even for training workloads, which typically exhibit stable batch processing, achieving full saturation is difficult due to distributed execution overheads. Training steps are frequently punctuated by non-compute operations, such as communication stalls in large-scale synchronization [29] and I/O blocking during weight checkpointing [63]. These recurrent idle periods, present in both inference and training, result in significant *temporal fragmentation* that static reservation policies cannot reclaim.

Memory underutilization. At the same time, memory underutilization is exacerbated by the growing diversity of AI models. Today's ML ecosystem spans massive LLMs with trillions of parameters [14, 35, 62] as well as smaller, specialized models designed for tasks such as OCR [66] or image generation [57]. Moreover, modern AI systems frequently compose these heterogeneous models into multi-stage pipelines to handle complex workloads. However, because existing sharing mechanisms often lack robust isolation, providers typically provision GPUs for the worst-case peak usage to prevent resource contention or out-of-memory (OOM) crashes. This often forces the assignment of an entire high-end GPU to a single model, resulting in severe memory fragmentation when the model consumes only a fraction of the capacity.

2.2 Existing GPU Sharing Mechanisms

Existing GPU-sharing mechanisms mainly aim to improve utilization but often compromise either isolation, elasticity, or both. As summarized in Table 1, these approaches generally fall into two categories: hardware partitioning and software-based sharing.

Hardware partitioning. Hardware-based approaches, such as NVIDIA MIG [47] and NVIDIA vGPU [49], partition a GPU into multiple fixed-size instances with dedicated SMs and device memory. This design offers strong compute, memory, and fault isolation with minimal runtime overhead. However, partition sizes are inelastic, and modifying them requires a GPU reset and the termination of all active applications.

	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.

Consequently, MIG cannot adapt to changing workload demands, which often results in resource fragmentation.

Software-based sharing. Software-based approaches handle GPU sharing at the application or runtime level. NVIDIA MPS [43] allows concurrent kernel execution but lacks mechanisms for memory management and fault isolation. More recent systems enhance compute sharing through improved scheduling strategies. For example, Tally [77] and LithOS [13] transform kernels to create sub-kernels that run on disjoint set of SMs for different tenants to improve SM utilization. However, because they operate at the application level and run transformed kernels in shared processes, a failure in one process can still propagate to others.

In contrast, GPreempt [17] and XSched [59] employ GPU-kernel-level preemption to support time-sharing, providing fault isolation but still neglecting memory as a shared resource. Without proper memory isolation, co-located workloads can interfere with one another through memory pressure and page fault contention, limiting the effectiveness of these software-based sharing mechanisms.

To introduce memory elasticity, TGS [68] enables GPU memory oversubscription by paging data to host memory. TGS employs UVM placement hints to prioritize memory allocation for interactive workloads than batch jobs. While TGS uses a "container" concept, it offers no guarantee or control over their memory usage or actual placement, which can lead to unpredictable stalls when multiple applications contend for limited memory resources (see §2.3).

2.3 Motivational Performance Study

To understand how existing GPU sharing mechanisms perform in real-world scenarios, we conducted an experiment by co-locating two representative workloads on an A100-40G GPU: latency-sensitive LLM inference (Llama 3.1 8B running on vLLM [34]) and best-effort image generation (Stable Diffusion 3.5 Medium [57] using the Diffusers framework [2]). This setup mimics real-world multimodal pipelines (*e.g.*, GPT-4o) where an LLM orchestrates visual generation tasks, which call both models for a single request.

Figure 2(a) shows the case where both workloads occupy 20GB of memory and fit within the GPU. Compared to exclusive execution, which we treat as ideal, the default NVIDIA driver time-shares GPU compute across processes. This causes vLLM's P99 TTFT and ITL[2] to increase by 10.4× and 2.7×, respectively, while diffusion throughput drops by 1.8×. The state-of-the-art GPU sharing system XSched [59],

[2]TTFT refers to *time to first token*, and ITL refers to *inter-token latency*. Both of them are key metrics for LLM inference latency.

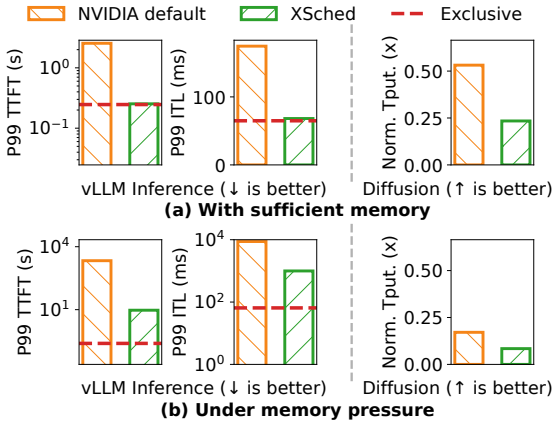


Figure 2: Performance of colocated latency-critical vLLM and a best-effort diffusion workloads on an A100-40G GPU. (a) With sufficient memory, XSched [59] preserves vLLM latency but reduces diffusion throughput. (b) Under memory pressure (managed by TGS [68]), both latency and throughput degrade sharply.

which implements user-space preemptive scheduling, preserves vLLM latency but reduces diffusion throughput to only 23.4% of ideal. This occurs even though vLLM consumes just 61.2% of SM time, leaving 38.8% of GPU compute idle.

This interference becomes even more severe under memory pressure. In production, vLLM typically reserves most of GPU memory for KV caches (*e.g.*, 36GB), yet actual memory occupancy fluctuates with request rates. Therefore, colocating steady best-effort workloads like Diffusion (20GB) can harvest the slack time during vLLM's low traffic periods. However, when vLLM's load ramps up, vLLM reclaims its reservation, and the combined footprint (56GB) now exceeds 40GB GPU capacity and requires paging. To manage swapping, we employ TGS [68], a state-of-the-art system for transparent GPU memory paging. Under these conditions, XSched's performance isolation deteriorates sharply: vLLM's P99 TTFT and ITL inflate to 37× and 15× of ideal, and diffusion throughput falls to 8.4% (Figure 2b). A deep investigation points to three main causes for this degradation:

- **Lack of memory isolation.** Figure 3 visualizes oscillating memory usage when vLLM and diffusion co-run under pressure. In stage #1, vLLM initializes and reserves most of the GPU memory, but soon get evicted when diffusion starts to allocate its memory and run (t=10–50s). At stage #3 (t=50s), vLLM starts receiving requests and competes for memory with diffusion, suffering from constant page thrashing and hence unpredictable latency spikes. Enforcing capacity isolation here would require precise control over device physical memory and page residency, which is unattainable for a user-space runtime.
- **Inefficient memory paging.** The extreme overall throughput loss of both vLLM and Diffusion is largely due to software overheads in swapping and fault handling. Our profiling reveals that the effective swap bandwidth is only 3.4 GB/s, just 21% of the 16 GB/s available PCIe bandwidth. As shown in Figure 4, this inefficiency stems from two sources: first, the driver's page eviction logic blocks the critical path for faulting in new pages, starving the host to device copy engine. Second, frequent page faults generate many small control messages between the host and device, preventing the driver from saturating the PCIe bus with application data. As a result, the CPU incurs a 3× overhead on handling page faults and sending small PCIe messages, while the blocking PCIe I/O further expand the diffusion execution time by 11×.
- **Inefficient compute scheduling.** Even without memory pressure, user-space kernel scheduling, exemplified by XSched, has to insert CUDA events to track GPU kernel execution status and conduct inter-process communication to coordinate co-running processes. However, these operations incur heavy blocking and overhead to GPU execution. Our profiling reveals that XSched, while preserving low latency for vLLM, only achieves 86.0% compute utilization and unnecessarily penalizes the best-effort diffusion task. In contrast, the default driver schedules processes directly and switches between them with hardware preemption support, which achieves 99.8% compute utilization.

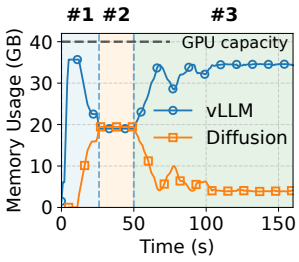


Figure 3: Memory contention visualization when total memory demand exceeds GPU capacity.

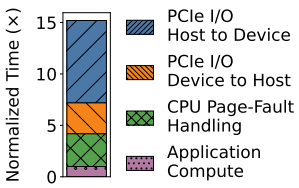


Figure 4: Diffusion time breakdown under memory pressure, normalized to it running with sufficient memory.

Takeaway. Existing GPU sharing systems implemented purely in user space cannot ensure efficiency and performance isolation. Without privileged access to GPU page tables, they cannot enforce process-level memory quotas, leading to unavoidable contention or OOM crashes. They also suffer from high scheduling overheads due to the lack of visibility into fresh kernel execution status. These findings suggest that efficient resource management requires lower-level hardware observability and control.

3 Dissecting GPU Resource Management

We ground the design of GVM in a detailed analysis of the open GPU driver [46]. Specifically, we studied the driver's source code to understand its internal architecture and interfaces, and we instrumented its compute and memory paths to uncover undocumented hardware mechanisms. The resulting insights shaped our design decisions in §4.

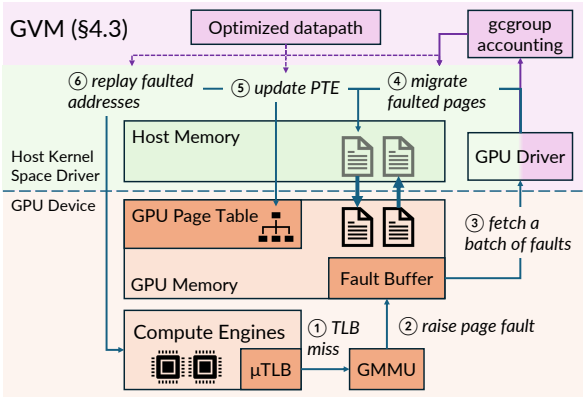


Figure 5: GPU hardware/software architecture for virtual memory management. The GPU MMU translates virtual addresses, and major page faults are handled by the driver in the host OS kernel.

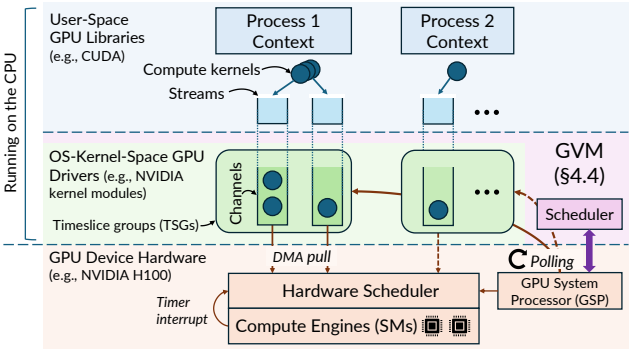


Figure 6: GPU compute scheduling hardware/software architecture.

3.1 GPU Virtual Memory

Modern GPUs support virtual memory in a manner analogous to CPUs. Beginning with NVIDIA's Pascal [41] 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, including the user-space 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) [5,40], 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 illustrated in Figure 5, 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 account for physical memory usage and classify 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 [48]).

3.2 GPU Compute Scheduling

GPU compute scheduling is a hierarchical process 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).

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 [9].

As Figure 6 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] [17].

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 C1: 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.

4 Design

4.1 GVM Overview

We developed GVM as a GPU virtualization layer in the GPU driver. Figure 7 presents an overview of GVM's design.

Abstraction (§4.2). GVM offers each process a container-like abstraction called a *GPU instance*. A GPU instance enforces strict limits on physical GPU resources while presenting the illusion of a private device. Applications run unmodified on a GPU instance and share the host OS and GPU driver, incurring minimal host overhead and process launching delay. Instances are managed via a `cgroup`-like filesystem interface, enabling standard orchestrators like Kubernetes [45] to dynamically adjust per-process compute and memory limits at runtime.

[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 $2 \times 44/1.5 \approx 59\mu s$.

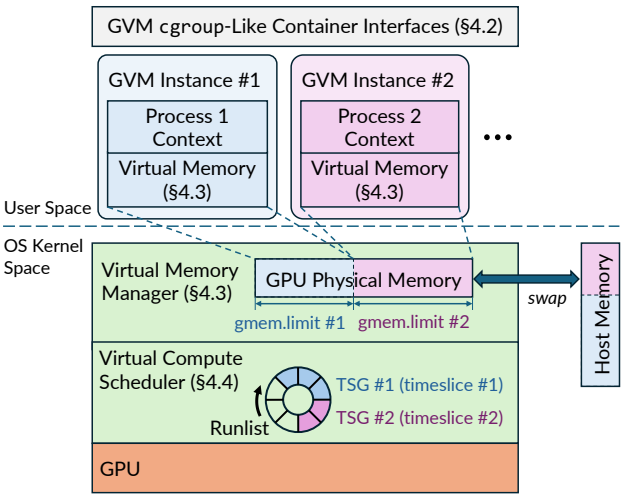


Figure 7: GVM design overview.

Virtual memory manager (§4.3). GVM defaults applications allocate virtual memory by intercepting and redicting CUDA memory management APIs to use UVM interfaces. Uniquely, GVM re-implements virtual memory management for two purposes. First, it enforces per-instance capacity isolation by instrumenting the physical page allocation path during fault handling. Second, GVM still transparently swaps pages out when an instance exceeds its quota to prevent out-of-memory errors. To keep temporary memory oversubscription and swapping efficient, GVM carefully reworks the driver page fault handling path to minimize PCIe round-trips and maximizes bi-directional bandwidth with overlapping.

Virtual compute scheduler (§4.4). GVM virtualizes the GPU compute by *time sharing* all GPU compute resources, including SMs, tensor cores, DMA engines, *etc.*, across running instances. GVM adopts a driver-hardware collaborative scheduling approach, where the GSP and GPU hardware scheduler still schedules kernels directly for efficiency, while GVM driver monitors application execution status and dynamically curates timeslices and TSGs, via privileged hardware RPCs, to `detach` low-priority GPU instances during contention and `attach` them to harvest idle cycles during gaps. This design ensures isolation and maximizes flexibility.

4.2 The GPU Container Abstraction

GVM 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.

Interface	Description
`/gmem.limit`	Maximum physical GPU memory (bytes)
`/gmem.current`	Current GPU memory usage (bytes)
`/hmem.limit`	Host memory quota for swapped pages (bytes)
`/compute.timeslice`	Max compute time slice (ms) per round
`/compute.priority`	Relative scheduling priority

Table 2: `gcgroup` interfaces for GPU resource limit.

`gmem.limit` enforces per-process GPU memory capacity. Once usage exceeds the limit, GVM transparently evicts pages to host memory. The optional `hmem.limit` bounds how much host memory can be used for swapped pages; setting it to zero disables swapping and falls back to raise explicit error when the process runs out of GPU memory. A GPU compute control group resembles a CPU cgroup but manages GPU execution time. A low-level `compute.timeslice` API specifies the maximum duration a process can occupy the GPU in a scheduling round, while a high-level `compute.priority` API sets its relative scheduling weight and preemption order. These parameters can be dynamically adjusted to implement priority-based or weighted-fair-share policies across processes (§4.4). GVM creates and manages all `gcgroup` directories inside the GPU driver. When a process initializes its CUDA context, GVM automatically instantiates metadata and exposes the control files. This lightweight, file-based interface enables compatibility with existing container infrastructures while allowing kernel-level enforcement of GPU isolation and elasticity.

4.3 Virtual Memory Management

Memory accounting and capacity isolation. GVM treats all GPU allocations as virtual and manages physical device memory as a shared pool partitioned across containers. Each container has a `gmem.limit` that bounds its device-resident working set and an optional `hmem.limit` that bounds how much host memory can be used for swapped pages.

When a batch of page faults arrives, GVM first accounts the new pages to the faulting container and checks its `gmem.limit`. If the container is within its limit, GVM proceeds to bring in the pages. If the container has exceeded its limit or its limit has been reduced, GVM 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 8).

To ensure accuracy, GVM 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 [42] and the driver tracks fine-grained mapping between each 64KB GPU page and its

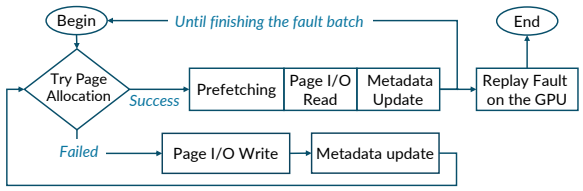


Figure 8: Baseline page-fault handling path in the stock NVIDIA GPU driver. The driver allocates device memory greedily and treats swapping as an exceptional, single-application event.

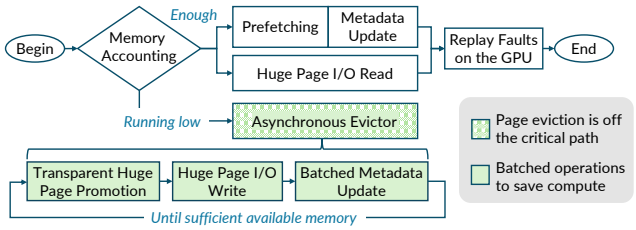


Figure 9: GVM's page-fault handling path. GVM enforces per-container limits, uses huge-page-aware batching, and decouples eviction from the critical fault path.

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. GVM 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, GVM now can opportunistically 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, GVM 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 8), leaving the PCIe, which supports concurrent bi-directional I/O, underutilized in most of the time. GVM therefore restructures the page-fault path to maximize throughput on both directions and minimize work on the critical path.

As shown in Figure 9, when handling a batch of page faults, GVM 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,

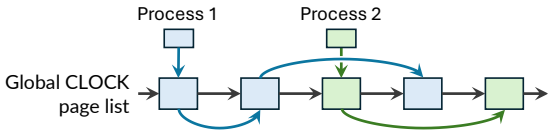


Figure 10: Dual-linked page list. Physical pages form a global CLOCK list, while each container maintains a secondary per-container chain for local eviction decisions.

GVM 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, GVM is able to 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 driver maintains a global CLOCK [11] list and evicts least-recent access pages by default. However, since GVM evicts pages per container, its eviction order differs from the global eviction order. To balance global memory pressure with per-container isolation, GVM maintains a dual-linked page list, shown in Figure 10. 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.

GVM 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 GVM enforce strict per-container capacity limits while still choosing cold victims, reducing thrashing and preserving throughput under oversubscription.

4.4 Virtual Compute Scheduler

GVM employs a hybrid scheduler that splits responsibilities between the driver and the hardware (Figure 11). Unlike prior systems like XSched [59], which interpose on every kernel launch and place scheduling logic on the critical path, GVM 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. GVM solves this by uncovering internal driver-firmware RPCs that manipulate the global runlist. These primitives allow the driver to instantly preempt low-priority workloads by shrinking their timeslices and exclude them from the runlist entirely when high-priority containers are active. This combination of hardware and driver enables both scheduling efficiency and policy flexibility.

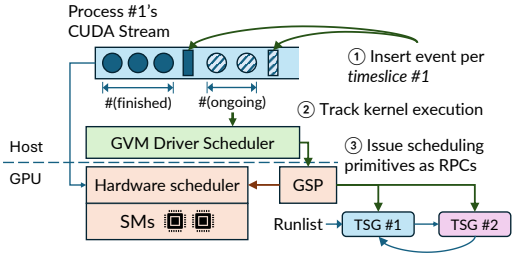


Figure 11: GVM's driver-hardware collaborative scheduling workflow. The GPU hardware schedules kernels directly for maximum efficiency, while the GVM driver dynamically adjusts the active runlist to enforce isolation policies.

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). GVM 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. The number of ongoing kernels is simply the difference between the launch count and the completion count.

Crucially, GVM uses these events in a novel way solely for *estimation*, not for driving the schedule directly. This distinction allows GVM to avoid the blocking synchronization required by prior user-space schedulers (§2.3). Instead, GVM 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, incurring negligible overhead.

Driver scheduling primitives. To make actual scheduling decisions, GVM 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. GVM uses this to force fine-grained preemption. By shrinking the timeslice of a running container, GVM compels the GSP to preempt running kernels once the timeslice expires.
- `DetachTSG`/`AttachTSG`: Removes or re-inserts a container's TSGs into the global hardware runlist. Detaching a TSG effectively suspends the container, ensuring it is completely excluded from consideration by the hardware scheduler.

Scheduling policies. Using these primitives, GVM 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. GVM 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, GVM supports weighted fairness, corresponding to fractional GPUs. Each container has a weight that encodes its target share. GVM 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.

5 Isolation

A common way to share GPUs today is to pass through an entire device to each OS container [33, 45, 68]. This provides strong isolation but no elasticity: once a container owns a GPU, its unused capacity cannot be reclaimed. GVM 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 GVM 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, GVM enforces per-container device memory limits (`gmem.limit`) and, when enabled, per-container swap usage. When a container exceeds its configured limit, GVM evicts that container's own pages to host memory instead of letting it consume unbounded device memory or trigger CUDA out-of-memory errors in other tenants. This gives each container a hard capacity cap plus optional oversubscription, and prevents one memory-hungry application from destabilizing others.

Temporal isolation of the full device. On the compute side, GVM's driver-resident scheduler (§4.4) 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, GVM 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, GVM 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 GVM.

5.2 Security Model and Limitations

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

Residual risks and comparison to hardware partitioning. GVM 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 GVM 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. GVM 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.

Limitations. GVM has several limitations. First, it requires only one high-priority application and will potentially penalize other colocated applications for performance isolation. 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, GVM does not offer spatial SM sharing to enforce strong isolation, but users are free to do so atop GVM with tools such as CUDA Green Context [39] and LithOS [13].

6 Implementation

GVM is implemented in NVIDIA open-source kernel modules with 5K lines of C code. GVM 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.

GVM 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, GVM implements a parallel interface over Linux `debugfs` that mimics the standard `cgroup` file-system structure.

GVM employs a thin user-space shim (similar to prior work [59, 68]) to hook CUDA calls. It redirects memory allocations (*e.g.*, `cuMemAlloc`) to GVM's modified UVM backend to enable transparent paging, while instrumenting kernel launches (*e.g.*, `cuLaunchKernel`) to track execution status. Inside the driver, GVM extends the existing UVM infrastructure, reusing native page tables while injecting custom logic for per-container accounting, isolation, and optimized bi-directional swapping.

7 Evaluation

Our evaluation aims to answer the following questions:

- Does GVM provide robust performance isolation and adaptively reallocate resources in a way that yields better performance than existing approaches? (§7.3)
- Can GVM flexibly trade-off latency slackness to best-effort throughput and stay on the Pareto frontier? (§7.4)
- What contributes to GVM's superior performance? (§7.5)

7.1 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 GVM against state-of-the-art prior work including TGS [68] (transparent memory oversubscription), XSched [59] (user-space scheduling), and GPreempt [17] (kernel-level preemption). We also evaluate industry-standard hardware partitioning using MIG [47].

TGS is the only baseline that natively supports memory 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 [43] 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.

7.2 Applications

We constructed two AI pipelines which cover three frameworks, three models with both LLM and diffusion models, training and inference, different priority levels, and different SLO requirements. These pipelines reflects emerging trends in multi-modal agentic systems and new continual learning workloads.

Multi-modal AI integrates the capability of LLMs and diffusion models to process and generate contents in both text and other modalities such as images and videos. We built a pipeline for this with a Llama-3.1 8B LLM and a stable-diffusion 3.5-medium diffusion model. The pipeline consists of a vLLM instance and a customized image generation instance built with Hugging Face diffusers library [2]. Since LLM inference is more latency-critical, it runs at a high priority. Diffusion requires a longer time for high-quality images with a loose or no latency requirement, so we treat it as a low-priority task. For this pipeline, we used BurstGPT trace [65] for LLM and VidProM dataset [64] for the diffusion model.

Continual online learning is an emerging LLM deployment paradigm that enables a model to keep learning from new data over time during inference [27, 72]. In a continual learning pipeline, an LLM is served for both inference and training with periodical weights update. To this end, we build such a pipeline for Llama-3.2-3B with vLLM [34] for inference and Llama-Factory [78] for LoRA finetuning. vLLM runs at a high priority to ensure low inference latency, while Llama-Factory serves as a best-effort task and runs only when vLLM cannot fully utilize the GPU. Following prior studies [75], we use the BurstGPT [65] trace for inference and the Alpaca dataset [61] for finetuning.

7.3 End-to-End Performance

We first compare GVM against all baselines in both memory-constrained and memory-sufficient settings.

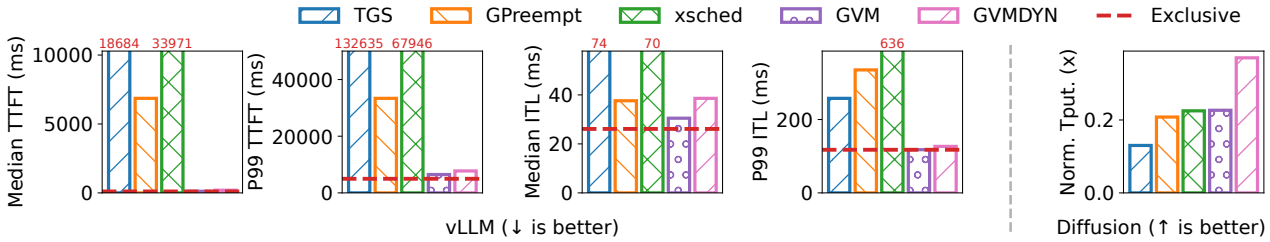
7.3.1 Performance Isolation

We run both pipelines in a memory-constrained configuration to test whether GVM can preserve high-priority performance while maintaining useful low-priority throughput. Results are shown in Figure 12.

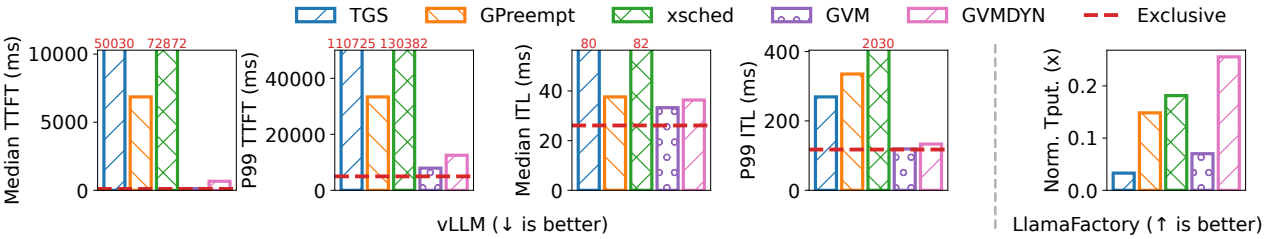
When co-locating vLLM (HP) and Diffusion (LP) (Figure 12a), GVM achieves the lowest vLLM latency and the highest diffusion throughput. Compared to the best-performing baseline, GVM reduces vLLM median and P99 TTFT and ITL by 59×, 5.1×, 1.2×, and 2.8×, respectively, while keeping all four metrics within 15% of the ideal latency achieved by vLLM running exclusively. At the same time, GVM delivers the highest diffusion throughput among all systems, demonstrating that strong isolation does not come at the cost of utilization.

All baselines experience high vLLM latency due to the lack of memory isolation. Similar to memory contention shown in Figure 3, 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 12b) also shows a similar trend. GVM reduces vLLM latency by 1.1-



(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 a high priority (HP) task and a low priority (LP) one under memory pressure. GVM achieves the lowest latency that is close to HP running exclusively, and high LP throughput.

53×, comparing against the best performing baseline. GVM does not achieve the best LP throughput in this case, because it has to enforce a strict memory limit on LlamaFactory to accommodate bursty vLLM load. Nevertheless, it still achieves the highest overall goodput.

Using the same `gcgroup`-like interface, we also implement a dynamic policy, GVMDYN, 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 12a and Figure 12b show, GVMDYN slightly increases vLLM latency relative to GVM that enforces strict isolation, but still outperforms all baselines. In return, GVMDYN improves LP throughput by an average of 2×, indicating that modest slack in HP latency can be traded for substantial LP throughput gains.

7.3.2 GPU Compute Efficiency

We next isolate the impact of GVM'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 13.

In this case, GVM achieves the lowest latency for vLLM (the high-priority task) across both pipelines. Specifically, GVM significantly outperforms MIG and achieves up to 480×, 183×, 2.75×, and 2.28× 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 GVM'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, GVM performs slightly worse than MIG but outperforms all other baselines. Compared to MIG, GVM achieves 86.7% Diffusion throughput for the multimodal AI workload and 88.3% 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.

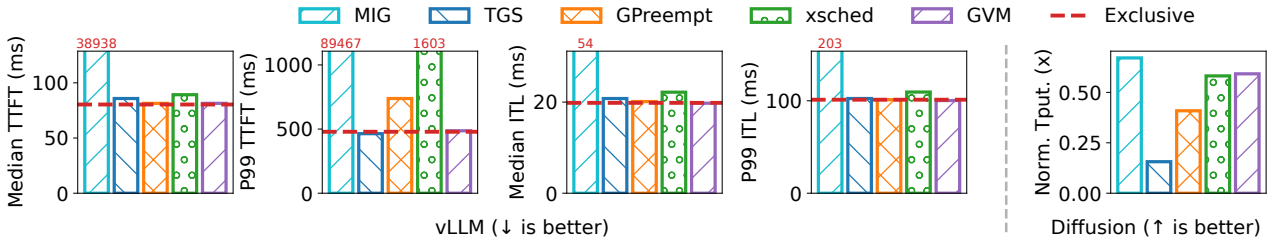
Overall, GVM achieves an ideal balance between HP and LP tasks by keeping consistently low HP latency while significantly improving LP throughput.

7.4 Dynamic Resource Coordination

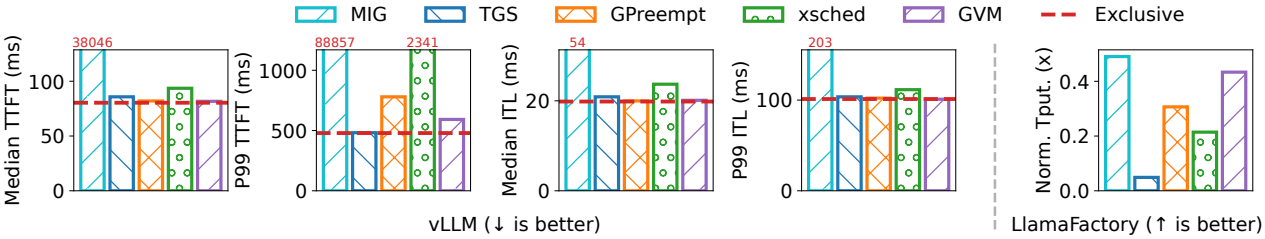
In this experiment, we evaluate GVM's elasticity by dynamically adjusting resource limit and allocation between applications and investigate whether can constantly achieve Pareto frontier compared to baseline systems.

Figure 15 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.



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



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

Figure 13: Performance variation when co-running vLLM with (a) Diffusion and (b) LlamaFactory when memory is sufficient.

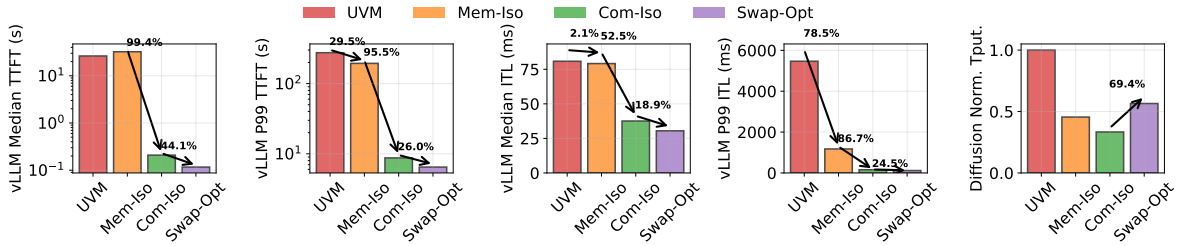


Figure 14: All three of GVM's contributions work in tandem to improve latency and throughput.

GVM, in contrast, achieves 100% SLO attainment while matching the best LP throughput among baselines. More importantly, GVM can adjust resource allocation according to HP latency requirements. By varying the compute duty cycle of the LP task, GVM trades off vLLM latency against diffusion throughput and eventually reaches full LP throughput. Across this range, GVM remains on the Pareto frontier, indicating that no other system dominates it on both metrics.

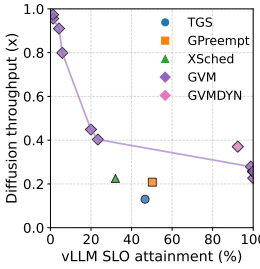


Figure 15: SLO attainment vs. throughput when co-locating vLLM (HP) and Diffusion (LP). GVM consistently stays on the Pareto frontier.

GVMDYN pushes this frontier further by dynamically reallocating GPU memory based on the current workload. It maintains over 90% SLO attainment for high-priority requests while achieving an LP throughput that static GVM can reach only when its SLO attainment drops to around 20%. This demonstrates the effectiveness of coordinated adjustment of both compute and memory limits in improving utilization.

7.5 Performance Analysis

We next investigate specific aspects of GVM's design to understand their individual contributions to overall performance.

Ablation Study. We incrementally enable GVM's mechanisms while co-locating vLLM and Diffusion under the same memory-constrained setting as in Figure 12. Figure 14 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 GVM'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, GVM removes this hidden source of interference.

Microbenchmarks. We further quantify the impact of GVM's swapping optimizations on low-priority tasks under

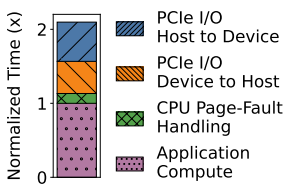


Figure 16: Time breakdown of Diffusion under memory pressure.

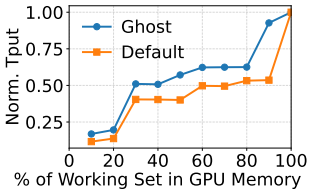


Figure 17: Normalized Diffusion throughput with regard to GPU memory usage.

memory contention. Figure 16 shows the time breakdown of diffusion when its available GPU memory is capped to 40% of its nominal requirement. With GVM, 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 4) due to synchronous blocking and small-packet PCIe congestion. Figure 17 plots diffusion throughput versus its GPU memory usage. Notably, GVM consistently outperforms the unoptimized baseline across all configurations, and can offload 10% of its working set with minor throughput drop.

8 Related Work

Software GPU partitioning. In addition to hardware partitioning solutions discussed in §2.2, systems such as HAMi [23], 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 that all workloads fit in device memory. In contrast, GVM 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 [38], Orion [60], and REEF [24], among others [19, 22, 53, 56, 69, 73], coordinate kernel launches across processes to improve utilization. Bless [76], Tally [77], and LithOS [13], along with others [9, 39, 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, 26, 44] 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 [8], SwapAdvisor [25], and Capuchin [55], as well as others [36, 37, 71], support memory offloading but rely on user-space control, provide weak isolation, and target specific workloads. Other GPU swapping systems [12, 30–32] improve prefetching or exploit faster interconnects, but require application modifications and still lack isolation. To our knowledge, GVM is the first system to provide both strong memory isolation and safe oversubscription for general GPU workloads.

9 Conclusion

GVM 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 GVM 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] 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.

[9] 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.

[10] G. Banga and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*, New Orleans, LA, Feb. 1999. USENIX Association.

[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] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for application-level resource management. *SIGOPS Oper. Syst. Rev.*, 29(5):251–266, Dec. 1995.

[17] 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.

[18] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating interference at microsecond timescales. In *OSDI*. USENIX Association, 2020.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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].

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

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

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

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

[38] 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.

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

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

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

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

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

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

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

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

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

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

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

[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] 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.

[64] W. Wang and Y. Yang. Vidprom: A million-scale real prompt-gallery dataset for text-to-video diffusion models, 2024.

[65] 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.

[66] H. Wei, Y. Sun, and Y. Li. Deepseek-ocr: Contexts optical compression. *arXiv preprint arXiv:2510.18234*, 2025.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] Y. Xu, Z. Mao, X. Mo, S. Liu, and I. Stoica. Pie: Pooling cpu memory for llm inference, 2024.

[72] 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.

[73] P. Yu and M. Chowdhury. Salus: Fine-grained gpu sharing primitives for deep learning applications, 2019.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.