Breaking the Tradeoff: Elastic and Isolated GPU Sharing with GVM

Yicheng Liu[*,‡], Yifan Qiao[*†,§], Tian Xia[§], Yi Xu[§], Tony Hong[§], Yutong Zhou[‡], David Sun[‡], Shuo Yang[§], Yilong Zhao[§], Junyi Shu[‡], Jiarong Xing[§], Harry Xu[‡], Ion Stoica[§], Joseph E. Gonzalez[§], Sam Kumar[‡]

[‡]UCLA [§]UC Berkeley

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 GVM, an OS-level GPU virtualization layer integrated into the open-source GPU driver. GVM 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 GPU page tables and extended 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 58× across three representative AI pipelines. GVM is open-sourced at https://github.com/ovg-project/GVM.

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 [55], at a cost of approximately $500k. Yet, modern AI workloads are increasingly diverse. Many specialized models are too small or lack sufficient compute or memory intensity to saturate such devices. As a result, major datacenters report GPU utilization typically remains below 30% for compute and below 60% for memory [14, 77].

One natural approach to improve utilization is to provide GPU virtualization and GPU sharing across multiple applications [14, 24, 43, 48, 52, 54, 66, 87]. This is particularly relevant for modern compound AI systems [56, 69, 70, 80, 85] and multi-agent frameworks [1, 17, 21, 60]. These systems consist of multiple distinct components that vary in their resource requirements. For instance, GPT-oss-120B [57] consumes ~65GB HBM, whereas smaller models like GPT-oss-20B [57] or Qwen3-Omni talker [80] require only 10–13GB. A single high-end GPU like H100 or B200 can, in principle, co-locate 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 [48] and recent research systems [14, 20, 24, 43, 66, 84, 87], 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 exceeds its memory capacity: a single process allocating excessive memory can trigger CUDA out-of-memory errors that crash colocated applications. Even if the system supports memory oversubscription [5, 45], 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 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 colocated workloads. Strong isolation can be obtained by using *hardware-level partitioning mechanisms* provided by GPU vendors, such as NVIDIA MIG [52] and vGPU [54], which statically divide GPU compute and memory resources. However, reconfiguring these partitions 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 modern AI workloads, building such a system is fundamentally challenging. Elastic resource allocation is driven by application behavior and runtime resource demand. Isolation, in contrast,

[*]Equal contributions, listed in alphabetical order.

[†]Corresponding author.

must be enforced below the application boundary through precise memory and compute accounting and controlled access to hardware resources, without relying on application cooperation or full knowledge of application semantics.

This challenge is compounded by the widening gap between *increasingly powerful GPU management capabilities* and the *outdated abstractions exposed to applications*. On one hand, modern GPUs provide rich control mechanisms for resource management, including preemption, 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 tradeoff: 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 [51] 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. Although our prototype targets NVIDIA, GVM depends on architectural primitives that also exist in AMD GPUs; §7.3 maps the vendor-specific scheduling, preemption, and fault-handling hooks needed for a port. 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 GVM, a novel GPU virtualization system integrated into the open-source GPU kernel driver [51]. GVM 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, GVM 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, GVM runs with OS kernel privileges, which enables precise memory accounting, enforced memory residency, and fault containment. In contrast to hardware partitioning, GVM achieves elasticity by continuously monitoring GPU state and adapting resource allocations within the driver, without requiring GPU resets or interrupting running applications. GVM provides a baseline level of elasticity and isolation entirely *transparently*, requiring no source code modifications. Additionally, it exposes an optional memory-pressure event interface that lets aware applications release and rematerialize semantic state to avoid swapping.

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, 45]. 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 design based on classical OS scheduling could trigger excessive remote procedure calls (RPCs) to the GPU across PCIe. To address this challenge, GVM 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, with full transparency, GVM can handle memory oversubscription only through swapping. Rematerializing disposable state, such as KV caches, often performs better but requires application involvement. GVM therefore exposes an optional `usermemfd` interface (§4.2), which reports `FREE` and `ALLOC` events and a target memory size to an application handler. The handler can release and later reconstruct application state without modifying model kernels or core logic. If no handler is provided or the handler cannot release enough memory, GVM falls back to swapping, preserving transparency and isolation at the cost of efficiency.

We have implemented GVM for NVIDIA GPUs and evaluated it on an A100 GPU using three representative AI

pipelines. Compared to the state of the art, GVM 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 [29], Meta [14], and Alibaba [77], report low GPU utilization. In an Alibaba production cluster [77], 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 [75]—and training is punctuated by communication stalls [30] and I/O blocking [72], 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 [15, 38, 68] to specialized models for OCR [76] and image generation [63], into multi-stage pipelines. These pipelines further exacerbate memory underutilization because the model state for every stage remains resident in GPU memory, even though only a subset of the stages is active at any given time.

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 [52], vGPU [54]) offers strong isolation but is inelastic—resizing requires a GPU reset and termination of all applications.

Software-based approaches trade isolation for flexibility. MPS [48] allows concurrent kernels but lacks memory and fault isolation. Tally [88] and LithOS [14] split kernels and run them in shared processes, so a fault in one can propagate to others. GPreempt [18] and XSched [65] employ kernel-level preemption for time-sharing and fault isolation but neglect memory as a shared resource. TGS [78] 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 colocate high-priority LLM inference (Llama 3.1 8B on vLLM [36]) with low-priority image generation (Stable Diffusion 3.5 Medium [63]) on an A100-40G GPU, mimicking real-world multimodal pipelines (*e.g.*, GPT-4o). To isolate the effects of SM sharing and swapping in this motivational study, we drive the applications with a synthetic request stream that maintains stable compute and memory pressure. The end-to-end evaluation in §6.3 instead uses a bursty production trace with the same application memory demands.

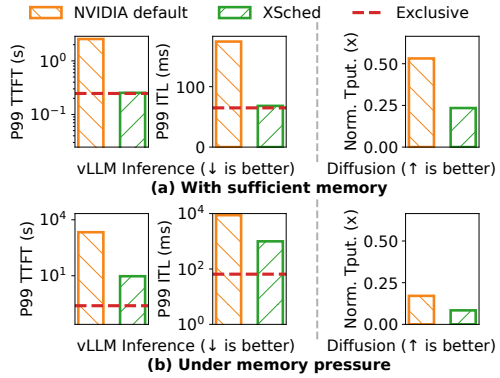


Figure 1: Performance of colocated vLLM and diffusion on an A100-40G GPU. (a) With sufficient memory. (b) Under memory pressure.

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 [65] preserves vLLM's latency but reduces diffusion's throughput to 23.4% of ideal, even though vLLM uses only 61.2% of SM time.

Under memory pressure—when vLLM reserves 36GB for KV caches and the combined footprint (56GB) exceeds GPU capacity—XSched with TGS [78] 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, colocated 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.

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

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

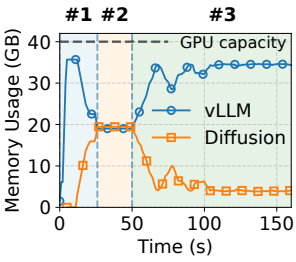


Figure 2: Memory contention visualization when total memory demand exceeds GPU capacity.

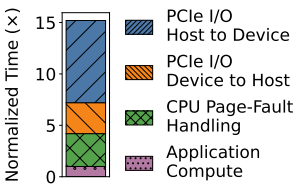


Figure 3: Diffusion time breakdown under memory pressure, normalized to it running with sufficient memory.

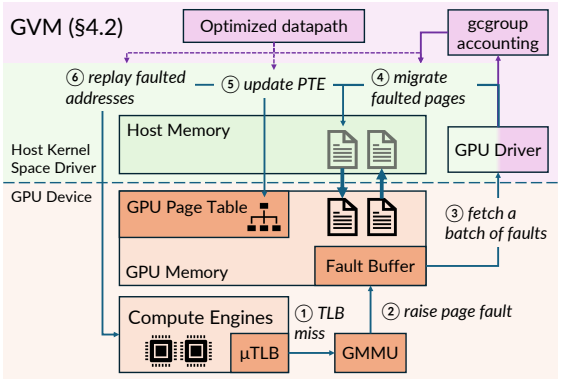


Figure 4: GPU hardware/software architecture for virtual memory management. The GPU MMU translates virtual addresses, and major page faults are handled by the driver in the host OS kernel.

3 Dissecting GPU Resource Management

We ground the design of GVM in a detailed analysis of the open GPU driver [51]. Specifically, we studied the driver's source code to understand its internal architecture and interfaces, and instrumented its compute and memory paths to uncover undocumented hardware mechanisms.

3.1 GPU Virtual Memory

Modern GPUs support virtual memory analogous to CPUs. Beginning with NVIDIA's Pascal [46] 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 kernel driver, which configures the GPU's MMU and handles page faults.

The NVIDIA GPU driver supports two distinct memory management models. Standard allocations using `cudaMalloc` are backed by statically mapped device pages that remain resident and are ineligible for paging. In contrast, Unified Virtual Memory (UVM) [5, 45] uses page faults to populate and migrate pages on demand. This allows GPU memory to be overcommitted and excess device-resident pages to be transparently evicted 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 RPCs to the device, including updating PTEs and initiating migration I/O. Frequent RPCs incur high CPU and PCIe overheads and throttle paging throughput (3.4GB/s as measured in §2.3). Moreover, this inefficiency will become increasingly critical as hardware bandwidth grows (*e.g.*, NVLink [53]).

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).

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 proces-

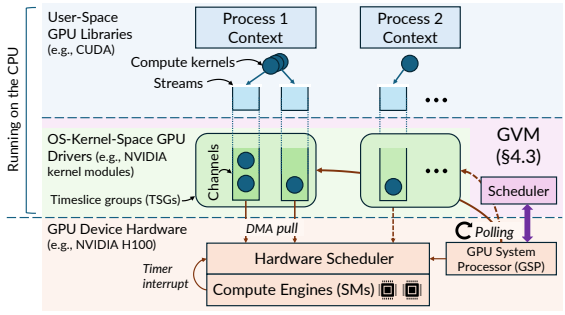


Figure 5: GPU compute scheduling hardware/software architecture.

sor (GSP), and the GPU hardware scheduler [11].

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. 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] [18].

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.

4 Design

Figure 6 presents an overview of GVM's design.

[3]An NVIDIA A100-40G GPU has 44MB of state (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 $2 \times 44/1.5 \approx 59\mu s$.

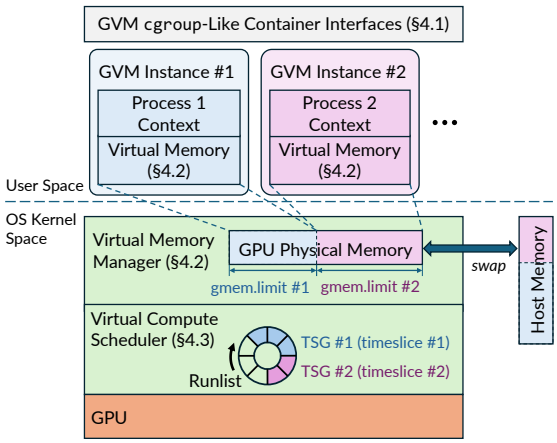


Figure 6: GVM design overview.

4.1 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.

`gmem.limit.high` enforces per-process GPU memory capacity. Once usage exceeds the limit, GVM transparently evicts pages to host memory; `gmem.limit.low` provides best-effort memory protection: GVM avoids swapping pages from a process whose `gmem.current` is at or below its `gmem.limit.low`, shielding it from eviction pressure caused by colocated 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). 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.

4.2 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.

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 timeslice (ms) per round
`/compute.freeze`	Freeze / unfreeze kernel execution
`/gcgroup.stat`	Application execution stats

Table 2: gcgroup interfaces for GPU resource management.

To enforce memory limits and decide when to swap to host memory, GVM must keep track of how much physical memory is allocated to each container. To achieve this, GVM 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 GVM a clean way to keep track of each container's memory usage.

When a batch of page faults arrives, GVM first accounts the new pages to the faulting container and checks its `gmem.limit.high`. 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 7).

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 [47] 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 sets, 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 the GMMU, GPU page faults are still raised 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 pages also simplify the GPU-host page mapping, and greatly reduce the CPU overhead to update this metadata. For small allocations or the remaining pages that are not aligned to a 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.

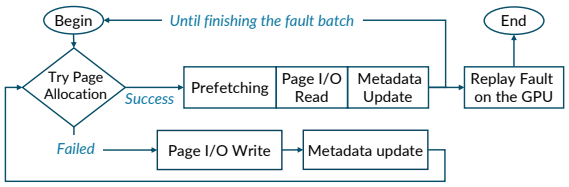


Figure 7: Baseline page-fault handling path in the stock NVIDIA GPU driver. The driver allocates device memory greedily and treats swapping as an exceptional, single-application event.

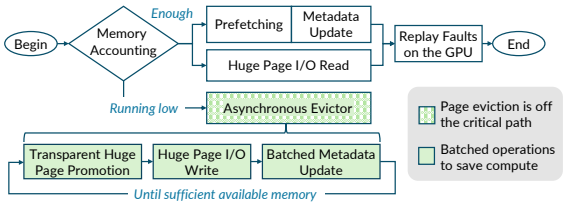


Figure 8: GVM's page-fault handling path. GVM enforces per-container limits, uses huge-page-aware batching, and decouples eviction from the critical fault path.

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

As shown in Figure 8, 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 triggered when a container's available memory budget is running low. Further, for the swap-in path, 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 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 driver maintains a global CLOCK [12] list, aiming by default to evict pages that are not recently accessed. 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 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.

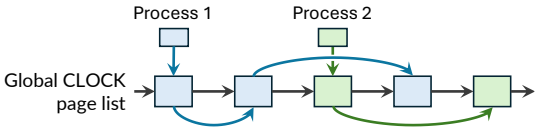


Figure 9: Dual-linked page list: physical pages form a global CLOCK list while each container maintains a secondary per-container chain.

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.

Application-system collaboration via memory pressure events. Although GVM heavily optimizes the paging datapath, transferring large working sets across the PCIe interconnect can still degrade performance. For many AI workloads, an efficient alternative to demand paging is *rematerialization* [27]—dropping intermediate values (*e.g.*, activation tensors during DNN training or KV cache 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. GVM resolves this through `usermemfd`, a file-descriptor-based notification interface conceptually similar to Linux's `userfaultfd`. An application creates a `usermemfd`, registers it with its GPU container, and runs an optional handler thread that waits with `poll()`. Each notification contains an event type and a target size in bytes.

GVM defines two events. A `FREE(size)` event indicates that memory pressure has increased and asks the application to release up to `size` bytes of rematerializable state. An `ALLOC(size)` event indicates that capacity has become available and allows the application to reconstruct or grow such state by up to `size` bytes. These events communicate *when* and *how much*; the application retains control over which semantic objects to drop or reconstruct. For example, our SAM handler, evaluated in §6.3.1, responds by decreasing or increasing the number of images processed in a batch. The required application change is only the registration and event handler; the model and its GPU kernels are unchanged.

The interface is advisory and non-blocking. On `FREE`, GVM continues enforcing the container's memory limit through transparent eviction, so an absent, slow, or partial response cannot stall reclamation or affect another tenant's isolation. A handler that releases memory promptly avoids the corresponding PCIe transfers; otherwise, the existing demand-paging path supplies the fallback. Similarly, allocations attempted after `ALLOC` remain subject to the container's current limit. Unmodified applications do not register a handler.

4.3 Virtual Compute Scheduler

GVM virtualizes the GPU compute by *time sharing* all GPU compute resources, including SMs, tensor cores, DMA engines, *etc.*, across running instances. GVM employs a hybrid scheduler that splits responsibilities between the driver and the hardware (Figure 10). Unlike prior systems like XSched [65], 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.

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 preempt low-priority workloads by shrinking their timeslices and excluding them from the runlist entirely when high-priority containers are active. This driver-hardware collaboration enables efficient and flexible scheduling.

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. Subtracting the completion count from the launch count gives the number of ongoing kernels.

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 the 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, 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.

5 Isolation

A common way to share GPUs today is to pass through an entire device to each OS container [35, 50, 78]. This provides strong isolation but no elasticity: once a container owns a

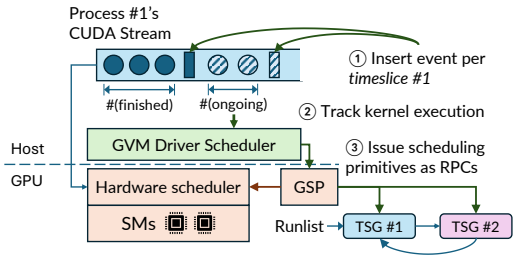


Figure 10: GVM's driver-hardware collaborative scheduling workflow. The GPU hardware schedules kernels directly for maximum efficiency, while the GVM driver dynamically adjusts the active runlist to enforce isolation policies.

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. For memory, GVM enforces per-container memory limits (`gmem.limit`) and, when enabled, per-container swap usage limits. 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 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 compute isolation. On the compute side, GVM's driver-resident scheduler (§4.3) allocates execution time across containers using priorities and weights. Unlike spatial partitioning, GVM does not reserve a fixed subset of SMs for each container. A scheduled container can therefore use all SMs, which lets a bursty, latency-sensitive workload exploit the full device instead of being confined to a small partition. Preemption and timeslice control determine when competing compute contexts may execute and enforce each container's configured compute timeslice. They do not, however, constitute independent reservations of every microarchitectural resource or guarantee identical kernel throughput across workload mixes.

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

Security model. We assume the host OS kernel, vendor GPU kernel driver, GVM 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. 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 protection boundary of standard OS processes: tenants cannot read each other's virtual address space or GPU memory, and software faults in one container do not crash another. Because GVM time-shares a monolithic GPU without flushing or partitioning all internal state, timing and cache-residency side channels may nevertheless remain. MIG spatially partitions SMs and portions of the memory system, reducing both interference and some side-channel surfaces, but confines each tenant to a rigid fraction of the device. GVM makes the complementary tradeoff: it preserves address-space, capacity, and temporal compute isolation while allowing compute time and memory to be reallocated elastically; deployments requiring stronger microarchitectural separation can use hardware partitioning.

6 Evaluation

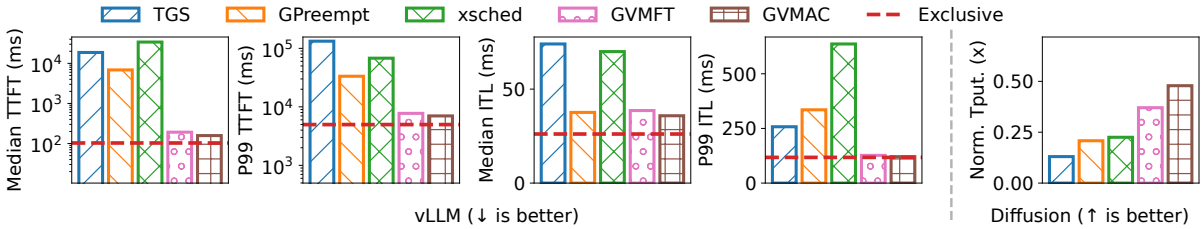
Our evaluation aims to answer the following questions:

- Does GVM provide robust performance isolation and adaptively reallocate resources in a way that yields better performance than existing approaches? (§6.3)
- Can GVM flexibly trade latency slack in a high-priority application for throughput in a low-priority application and stay on the Pareto frontier? (§6.4)
- What contributes to GVM's superior performance? (§6.5)

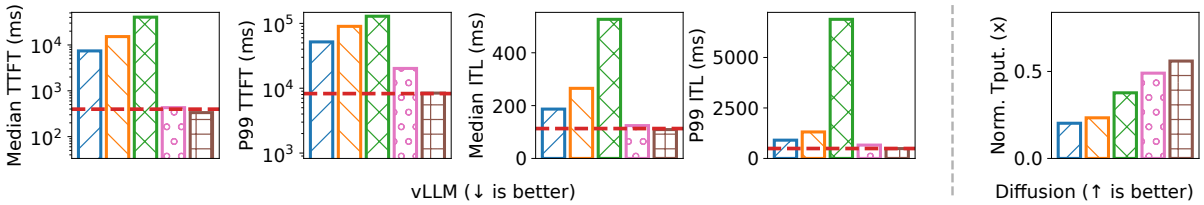
6.1 Setup

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.

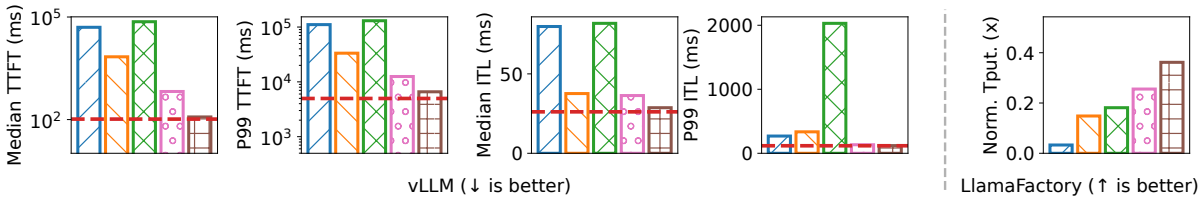
Orchestrator policy. We run a workload-specific policy in a userspace orchestrator process for one high-priority (HP) and one low-priority (LP) application (§A.2). It learns the HP application's demand from generic kernel-progress counters and limits or preempts the LP application when HP slowdown or backlog indicates contention. The policy does not inspect model or framework state and requires no workload-specific interface: the userspace process interacts with GVM only through its cgroup-style statistics and resource controls. The policy logic therefore remains in user space, while GVM's GPU-driver mechanisms enforce the requested limits and preemptions. Other workload mixes



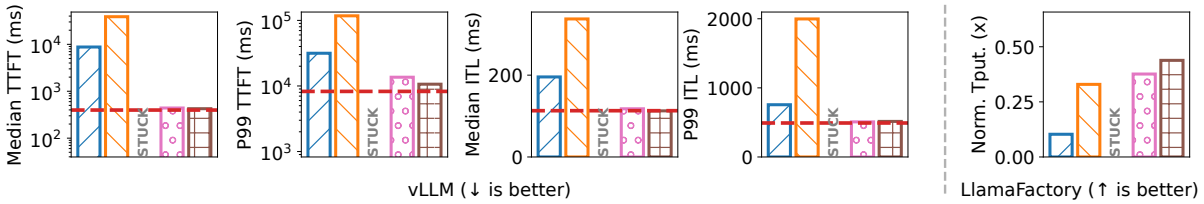
(a) Multimodal AI (Text Pipeline): vLLM (high priority) + Diffusion (low priority).



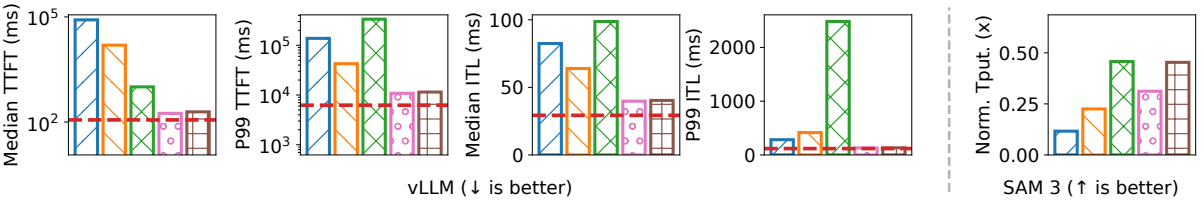
(b) Multimodal AI (Video Pipeline): vLLM (high priority) + Diffusion (low priority).



(c) Continual online learning (Text Pipeline): vLLM (high priority) + LLaMA-Factory (low priority).



(d) Continual online learning (Video Pipeline): vLLM (high priority) + LLaMA-Factory (low priority).



(e) Image detection: vLLM (high priority) + SAM (low priority).

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

can use different policies over the same interface.

GVM configurations. We evaluate three configurations. GVM is the underlying mechanism with statically configured `gcgroup` resource limits and no orchestrator. GVMFT (fully transparent) adds the orchestrator above, which dynamically adjusts the LP application's resources using only `gcgroup` statistics and controls. GVMAC (application collaborative) adds the optional `usermemfd` rematerialization handler to GVMFT (§4.2). Thus, AC additionally uses application semantics to rematerialize state and avoid swapping.

Baselines. We compare GVM against state-of-the-art prior work including TGS [78] (transparent memory oversubscription), XSched [65] (user-space scheduling), and GPreempt [18] (kernel-level preemption). We also evaluate industry-standard hardware partitioning using MIG [52].

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. Note that TGS builds upon MPS [48] and UVM optimizations, serving as a strictly stronger baseline than standard MPS. We have verified that TGS consistently achieves the same or higher performance than MPS.

6.2 Applications

We constructed three AI pipelines covering three frameworks, multiple models (LLM, VLM, and diffusion models

for video and image processing), training and inference, and different priority/SLO requirements. These pipelines reflect trends in multimodal agents and continual learning.

Multimodal AI colocates a high-priority LLM with a low-priority diffusion, replicating the resource contention in omni-modal serving frameworks [69, 70]. The text variant pairs Llama-3.1 3B (vLLM [36], high priority) with Stable Diffusion 3.5 Medium (Diffusers [2], low priority); the video variant uses Qwen2.5-VL-3B [9] for video-to-text inference and WAN2.1-Small [73] for video generation. We use BurstGPT [75] for LLM/VLM and VidProM [74] for diffusion.

Continual online learning serves an LLM for both inference and LoRA fine-tuning with periodic weight updates [28, 82]. The text variant pairs Llama-3.2-3B on vLLM [36] (high priority) with LLaMA-Factory [90] for fine-tuning (low priority); the video variant uses Qwen2.5-VL-3B [9] for both serving and training. Following prior studies [86], we use BurstGPT [75] for inference, Alpaca [67] for text fine-tuning, and MMVU [89] for video fine-tuning.

Image detection colocates high-priority vLLM serving with low-priority Segment Anything Model (SAM) image detection on COCO[37]. For GVMAC, SAM adjust its batch size when GPU memory is reallocated between the applications.

6.3 End-to-End Performance

6.3.1 Performance Isolation. We run all three pipelines in a memory-constrained configuration to test whether GVM can preserve high-priority performance while maintaining useful low-priority throughput. For the text multimodal pipeline, the application memory demands match those in our motivational setup. Figure 1b shows the motivational experiment, which uses a synthetic request stream to maintain stable pressure, whereas this experiment uses the BurstGPT trace described above to reflect production arrival patterns. We evaluate the GVMFT and GVMAC configurations defined above. Results are shown in Figure 11. When colocating vLLM (HP) and Diffusion (LP) in the text pipeline (Figure 11), GVMAC achieves both near-exclusive vLLM latency and the highest diffusion throughput among all systems. Compared to the best-performing baseline, GVMAC 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. GVMFT achieves comparable HP isolation while trading modest latency for improved LP throughput. The video pipeline shows a similar trend: GVMAC 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.

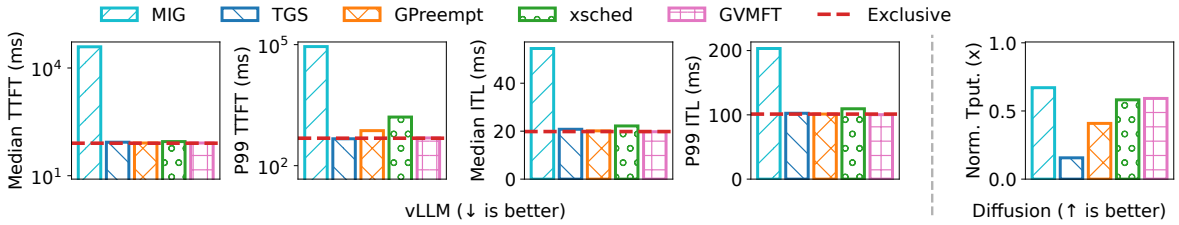
Colocating vLLM and LLaMA-Factory (Figure 11) shows a similar trend. GVMAC achieves vLLM latency within 16% of exclusive while delivering the highest LP throughput, outperforming the best baseline by 2×. GVMFT trades HP latency for further LP throughput gains. In the video pipeline, XSched becomes stuck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

In GVMFT, the orchestrator uses the same `gcgroup` interface to dynamically relax the LP task's memory limit when vLLM has a low load that cannot fully use its reservation. As Figure 11 shows, GVMFT slightly increases vLLM latency relative to static GVM, but still outperforms all baselines. In return, GVMFT improves LP throughput by an average of 2×, indicating that modest slack in HP latency can be traded for substantial LP throughput gains.

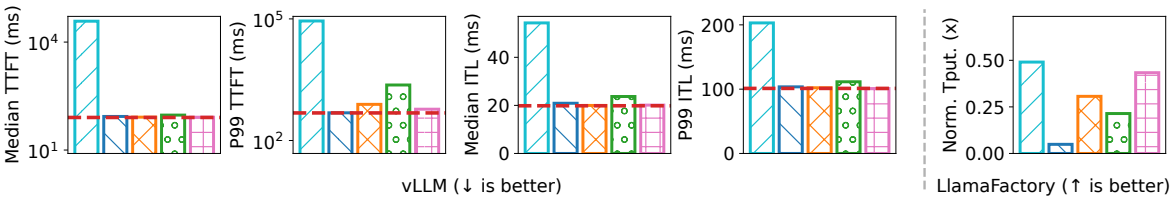
The image detection pipeline shows a similar trend. As Figure 11 shows, GVMAC provides the best overall balance between vLLM latency and SAM throughput. SAM has relatively low compute intensity so its throughput scales well with larger batches. GVMAC exploits this property by increasing SAM's batch size when more memory is available and reducing it when vLLM needs the memory. It therefore improves SAM throughput over GVMFT while preserving similar vLLM isolation; baselines that obtain comparable SAM throughput incur much higher vLLM latency.

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

In this case, GVMFT achieves the lowest latency for vLLM (the high-priority task) across both pipelines. Specifically, GVMFT 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 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, GVMFT performs slightly worse than MIG but outperforms all other baselines. Compared to MIG, GVMFT achieves 88% Diffusion throughput for the multimodal AI workload and 88% LLaMA-Factory throughput for the continual learning workload. MIG achieves the best LP throughput because it allocates the remaining 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



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



(b) Continual online learning: vLLM (high priority) + LLaMA-Factory (low priority)

Figure 12: Performance variation when colocating vLLM with (a) Diffusion and (b) LLaMA-Factory when memory is sufficient.

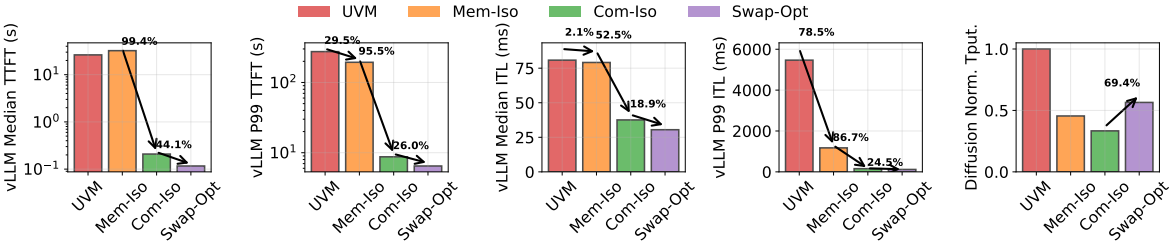


Figure 13: All three of GVM's contributions work in tandem to improve latency and throughput.

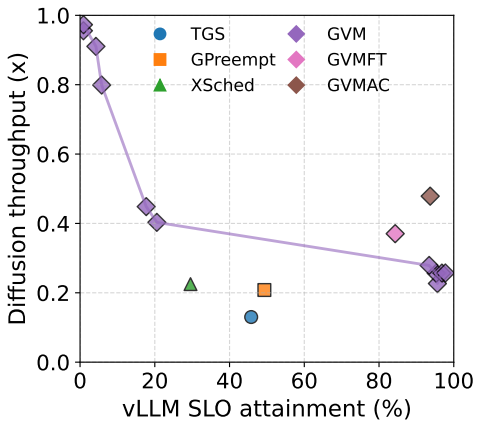


Figure 14: vLLM SLO attainment vs. diffusion throughput.

enough isolation to keep vLLM latency low.

Overall, GVMFT 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 GVM by adjusting resource limits between applications and investigate whether it remains on the Pareto frontier relative to baseline systems.

Figure 14 plots SLO attainment against diffusion throughput when colocating 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.

Static GVM, in contrast, achieves 100% SLO attainment while matching the best LP throughput among baselines. To obtain its curve, we vary static `gcgroup` configurations, including the LP task's compute duty cycle, across runs. These configurations trade vLLM latency against diffusion throughput and eventually reach full LP throughput. Across this range, static GVM remains on the Pareto frontier, indicating that no other system dominates it on both metrics.

GVMFT and GVMAC push this frontier further outward by dynamically reallocating GPU memory based on the current workload. GVMAC maintains over 93% SLO attainment while achieving 0.48× normalized LP throughput—a level that static GVM 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 GVM's mechanisms while colocating vLLM and Diffusion under the same memory setting as in Figure 11. Figure 13 shows the results.

Adding memory isolation alone protects vLLM from heavy

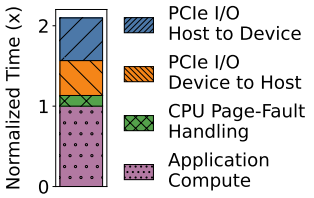


Figure 15: Diffusion time breakdown under memory pressure.

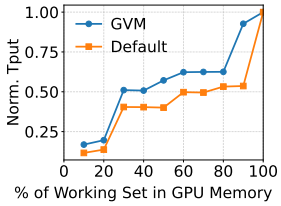


Figure 16: Diffusion throughput vs. memory usage.

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 the throughput of Diffusion, which previously suffered from slow swapping, by 69.4%. 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 largely mitigates this hidden source of interference.

Microbenchmarks. We further quantify the impact of GVM'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 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 3) due to synchronous blocking and small-packet PCIe congestion. Figure 16 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 only a minor throughput drop.

6.6 GPU Memory Management Techniques

We isolate the benefits of GVM's GPU memory management techniques using Diffusion. We limit Diffusion's device memory to 6 GB and cumulatively add asynchronous bidirectional swapping, dual CLOCK lists, and transparent GPU huge pages to the base implementation. We report throughput normalized to an exclusive.

Figure 17 shows that each technique improves Diffusion throughput under memory pressure. Asynchronous swapping allows paging to overlap with useful work, dual CLOCK lists further improve victim selection under the container limit, and transparent GPU huge pages reduce the overhead of moving and managing swapped pages. Together, these techniques substantially recover the performance lost to memory pressure.

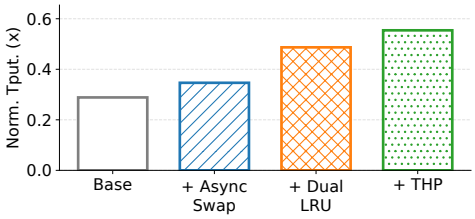


Figure 17: GVM's GPU memory management techniques.

7 Discussion

7.1 Scheduling Rounds

The GPU firmware performs round-robin scheduling for the TSGs attached to its runlist; a scheduling round is determined by the runnable TSGs and their configured compute timeslices rather than by a fixed application-level interval. A longer timeslice lets a TSG run longer before a context switch, whereas a shorter one improves responsiveness at the cost of potentially more context switches. A policy can change these timeslices at runtime through `gcgroup`. Memory eviction is independent of the scheduling-round duration: the memory manager selects pages using its per-container and global CLOCK lists when a memory limit or device pressure requires reclamation.

7.2 Microarchitectural Isolation

GVM does not partition the shared L2 cache, HBM channels and bandwidth, copy engines, interconnects, memory-controller queues, or other internal GPU structures. Fine-grained scheduling reduces simultaneous compute contention because only eligible contexts receive SM time, but state left in shared caches and fabrics can affect the next timeslice. Asynchronous memory transfers may also contend for shared bandwidth independently of SM scheduling. Consequently, GVM enforces per-container compute timeslices and memory-capacity limits, but does not guarantee invariant per-kernel performance under an adversarial workload that stresses these unpartitioned resources. Users requiring stronger separation can add spatial compute allocation with mechanisms such as CUDA Green Context [44] or LithOS [14], or choose a hardware-partitioned deployment.

7.3 Portability beyond NVIDIA

GVM's implementation is NVIDIA-specific, but its design requires three architectural capabilities rather than NVIDIA TSGs or UVM specifically: (1) preemptible, time-multiplexed execution contexts whose eligibility or share the driver can control; (2) recoverable GPU page faults handled by the host driver, which can update mappings before replay; and (3) a programmable kernel driver in which to add per-container accounting and policy.

AMD GPUs provide corresponding mechanisms in a different software stack. The AMD GPU Hardware Scheduler supports queue oversubscription, while newer GPUs use the MicroEngine Scheduler (MES) to map, unmap, prioritize,

and time-share user queues [39, 40]. Compute Wave Save and Resume (CWSR) permits preemption during a compute wave, and XNACK retry faults let the host driver service recoverable faults identified by process address-space ID and virtual address [39]. Thus, an AMD port would replace GVM's TSG/runlist RPC layer with KFD/MES queue-control hooks, its NVIDIA UVM hooks with AMD GPU/KFD fault and migration paths, and its CUDA interception shim with the corresponding HIP allocation and launch interfaces. The container-level accounting, limit enforcement, eviction policy, and scheduling policy would remain unchanged.

The AMD stack also exposes mechanisms that could remove some compromises of the NVIDIA implementation. Per-queue CU masks can restrict a queue to selected compute units [8], enabling an AMD implementation to combine GVM's temporal elasticity with optional spatial allocation. Direct access to the open-source AMD GPU/KFD scheduling and memory-management paths could also move policy hooks closer to queue management and fault handling, rather than communicating through NVIDIA's proprietary GSP interface.

7.4 Limitations

GVM relies on the user or operator to set and adjust each container's parameters, such as memory limits and compute priorities. Our evaluation focuses on preserving one high-priority application's performance under contention. Although GVM supports other settings, it does not decide which application to prioritize when resources are scarce; this is a policy decision for the orchestrator.

Virtual memory and on-demand physical-page allocation add overhead on the first access to newly allocated virtual addresses. This occurs primarily during initialization, when allocations are frequent, and does not affect later kernel execution for workloads that allocate infrequently.

8 Related Work

Software GPU partitioning. In addition to hardware partitioning solutions discussed in §2.2, Gdev [33] pioneered OS-level GPU resource management targeting Fermi-era GPUs with APIs for device memory swapping and logical GPU partitioning. However, it lacked preemptive execution and required application rewriting. Later systems such as HAMi [23], run:ai [64], 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, GVM virtualizes GPUs inside the kernel driver to support dynamic reallocation and strong isolation for unmodified applications with modern page-fault and preemption mechanisms.

GPU kernel scheduling. Many GPU kernel schedulers have been proposed for efficient compute sharing. Systems such as Paella [43], Orion [66], Reef [24], and others [19, 22, 59, 62, 79, 83], coordinate kernel launches across processes to improve utilization. Bless [87], Tally [88], and LithOS [14], others [11, 44, 58], 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, 49] 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 [10], SwapAdvisor [25], Capuchin [61], and others [41, 42, 81], support memory offloading but rely on user-space control, provide weak isolation, and target specific workloads. Other GPU swapping systems [13, 31, 32, 34] 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.

Confidential GPU computing. Graviton [71] proposes hardware-backed secure GPU contexts that isolate application code and data from other GPU contexts and from privileged host software, including the driver, OS, and hypervisor. NVIDIA's confidential-computing GPUs [16] use an on-device root of trust, secure boot, attestation, protected device memory, and an encrypted channel to a confidential VM to provide confidentiality and integrity for data in use. These systems address a stronger confidentiality threat model than GVM, but do not provide its cgroup-style elastic resource management. Their protection mechanisms are complementary to GVM's compute and memory isolation among GPU-sharing applications.

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.

10 Acknowledgment

We thank the anonymous reviewers for their comments and are particularly grateful to our shepherd for his/her valuable feedback. This work was supported by the National Science Foundation (CNS-2301343, CNS-2330831, CNS-2403254, IIS-2546642, IIS-2551122), the Alibaba Research Intern Program, and gifts to Sky Computing Lab from industry partners (Accenture, AMD, Anyscale, Cisco, Google, IBM, Intel, Intesa Sanpaolo, Lambda, Lightspeed, Mibura, Microsoft, NVIDIA, Samsung SDS, and SAP).

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] AMD. Setting the Number of Compute Units. https://rocm.docs.amd.com/en/latest/how-to/setting-cus.html, 2026. [Accessed 03-09-2026].

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] G. Dhanuskodi, S. Guha, V. Krishnan, A. Manjunatha, M. O'Connor, R. Nertney, and P. Rogers. Creating the first confidential GPUs: The team at NVIDIA brings confidentiality and integrity to user code and data for accelerated computing. *ACM Queue*, 21(4):68–93, 2023.

[17] 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.

[18] 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.

[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] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-Class GPU resource management in the operating system. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 401–412. USENIX Association, 2012.

[34] 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.

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

[36] 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.

[37] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár. Microsoft coco: Common objects in context, 2015.

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

[39] Linux Kernel Documentation. AMDGPU Driver. https://docs.kernel.org/gpu/amdgpu/module-parameters.html, 2026. [Accessed 03-09-2026].

[40] Linux Kernel Documentation. AMDGPU MicroEngine Scheduler (MES). https://docs.kernel.org/gpu/amdgpu/gc/mes.html, 2026. [Accessed 03-09-2026].

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

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

[43] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-resolution image synthesis with latent diffusion models, 2021.

[64] Run:ai. Quickstart: Launch Workloads with GPU Fractions. https://docs.run.ai/v2.17/Researcher/Walkthroughs/walkthrough-fractions/, 2023. Accessed: August, 2025.

[65] 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.

[66] 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.

[67] 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.

[68] K. Team. Kimi k2: Open agentic intelligence, 2025.

[69] S.-O. Team. Sglang-omni. https://docs.sglang.io/sglang-omni/, 2026.

[70] vLLM Omni Team. vllm-omni. https://docs.vllm.ai/projects/vllm-omni/en/latest/, 2026.

[71] S. Volos, K. Vaswani, and R. Bruno. Graviton: Trusted execution environments on GPUs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 681–696. USENIX Association, 2018.

[72] 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.

[73] 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.

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

[75] 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.

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

[77] 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.

[78] 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.

[79] 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.

[80] 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.

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

[82] 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.

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

[84] S. Yu, Y. Qiao, M. Ma, Y. Li, S. Yang, X. Tong, Y. Wang, Z. Xie, Y. An, S. Cao, K. Bao, D. Vij, X. Ding, Y. Wang, Q. Lu, Z. Wang, G. Gao, H. Xu, J. Shu, J. Xing, and Y. Sheng. Prism: Cost-Efficient Multi-LLM serving via GPU memory ballooning. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pages 75–93, Seattle, WA, July 2026. USENIX Association.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] Y. Zhao, L. Xie, H. Zhang, et al. Mmvu: Measuring expert-level multi-discipline video understanding. *arXiv preprint arXiv:2501.12380*, 2025.

[90] 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 GVM by extending NVIDIA open-source kernel modules with 5K lines of C code. GVM currently supports NVIDIA GPU architectures newer than Pascal.

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 [65, 78]) to hook CUDA calls. It redirects memory allocations (*e.g.*, `cuMemAlloc`) to GVM's modified UVM backend (§4.2) 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.

A.2 Scheduling Policies

GVM exposes workload-agnostic mechanisms through its cgroup-style `gcgroup` interface; scheduling policy remains in user space. This separation is intentional: a useful scheduler reflects its workload's service objective and demand pattern, but should not require workload-specific driver hooks or framework APIs.

Evaluated workload-specific policy. Our evaluation uses one high-priority (HP) application and one low-priority (LP) application. The scheduler reads only generic `gcgroup.stat` counters, including counts of submitted, completed, and pending kernels, and changes only the LP memory limit and compute state. It learns the HP application's clean completion behavior online, classifies its demand as idle, light, or heavy, and moves the LP application among *unlimited*, *memory-limited*, and *preempted* states. The LP application is restricted when the HP application exceeds its slowdown budget or develops a persistent backlog, and is given more resources when the HP application recovers or becomes idle.

The operator supplies the HP and LP memory limits, sampling and idle intervals, and the acceptable HP slowdown; the remaining thresholds are learned from execution. Thus, the policy is specific to the HP/LP objective, while its implementation uses only GVM's standard resource-control interface. A scheduler for another workload mix can express a different policy over the same files without changes to GVM or to the applications.

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.

B Artifact Appendix

B.1 Abstract

The artifact contains the GVM kernel module and CUDA interception layer, the GVMFT scheduler, the GVMAC-enabled vLLM implementation, three colocation workloads, baseline systems, experiment scripts, and plotting tools. It reproduces the paper's evaluation of high-priority vLLM inference colocated with low-priority diffusion inference, LLaMA-Factory fine-tuning, or SAM image detection. Because the artifact requires a modified NVIDIA kernel module and a server with a GPU, we provide evaluators with a server on which all software, models, and data are preinstalled.

B.2 Artifact check-list (meta-information)

- **Algorithm:** GPU memory isolation and priority scheduling; application-cooperative scheduling.
- **Program:** vLLM, Diffusers, LLaMA-Factory, and SAM; GVM, GVMFT, GVMAC, XSched, TGS, and GPreempt.
- **Compilation:** GCC, NVCC, CMake, Ninja, and Python 3.12; the evaluation server is prebuilt.
- **Models:** Llama-3.2-3B and Stable Diffusion 3.5 Medium for text; Qwen2.5-VL-3B and WAN2.1-Small for video; SAM 3 for image detection; model weights are cached on the evaluation server.
- **Data set:** BurstGPT request trace, COCO image-detection data, and the prompt/training data cached with the artifact.
- **Run-time environment:** Ubuntu Linux, CUDA 12.9, NVIDIA driver 575.57.08, Docker, and root access for kernel/debugfs operations.
- **Hardware:** NVIDIA A100-40 GB for most evaluations; A100-80 GB only for the no-memory-limit evaluation.
- **Execution:** each run lasts 600 seconds plus model startup and shutdown.
- **Metrics:** vLLM TTFT and inter-token latency distributions; diffusion throughput; training throughput; SAM image-detection throughput; SLO attainment.
- **Experiments:** overall performance, no-memory-limit, and Pareto evaluations driven by `exp/run_all.sh`.
- **Preparation time:** under 1 minute for remote access and the basic test; the software is preinstalled.

- **Publicly available:** https://github.com/easonycliu/gvm.
- **Code licenses:** component-specific open-source licenses included in the repository.
- **Data licenses:** BurstGPT and model-specific upstream licenses.
- **Workflow automation:** Bash and Python scripts.
- **Archived:** 10.5281/zenodo.21926635.

B.3 Description

B.3.1 How to access. The source artifact is available at https://github.com/easonycliu/gvm. Evaluation uses an author-provided server because GVM requires a modified NVIDIA kernel module and direct access to a supported GPU. An evaluator should send only their SSH *public* key to the AE chairs through the private AE channel. The author will send back instructions on how to connect to the server through the private AE channel. The remote-access procedure follows the ASPLOS allowance for artifacts requiring specialized hardware.

B.3.2 Dependencies. The provided server contains the required GPU, modified driver, CUDA toolkit, Python environments, Docker image, data, cached model weights, and repository checkout. Evaluators need only an SSH client and the private key corresponding to the public key they submitted. The repository `README.md` records the complete hardware, software, model, data, and from-scratch installation details.

B.4 Installation and basic test

No installation is required on the evaluation server. After logging in, run:

```
cd ~/gvm/exp
./run_check.sh
```

The test checks the modified kernel interface, all application environments, the adapted vLLM source installation, and a 1 GiB CUDA allocation through the GVM interception layer. It succeeds when every line is marked `PASS`, the allocation reports `total cuda memory allocated: 1024MB`, and the last line is `All checks passed.` See the README's `Kick-The-Tire` section for the complete expected output.

B.5 Experiment workflow

Start with the README's evaluation section. For each paper configuration, the evaluator enters `exp/vllm+diffusion`, `exp/vllm+llama-factory`, or `exp/vllm+sam` and invokes `../run_all.sh`. The `-method` argument selects GVM, GVMFT, GVMAC, or a baseline, while `-eval` selects the overall, no-memory-limit, or Pareto experiment. The script starts both applications, runs the 600-second request trace, shuts them down, and writes timestamped results under `data/`. Exact commands, output naming, parameter sweeps, and troubleshooting remain in the README.

B.6 Evaluation and expected results

The artifact reproduces paper figures and tables. The overall evaluation should reproduce the paper's relative comparison among GVM, GVMFT/GVMAC, and the baseline systems. The Pareto sweep should reproduce the reported tradeoff between vLLM SLO attainment and LP throughput.