Safe and Programmable OS-Level GPU Resource Management with eBPF

Yusheng Zheng*
University of California, Santa Cruz
Santa Cruz, United States
yzhen165@ucsc.edu

Tong Yu*
eunomia-bpf Community
Wuhan, China
yt.xyxx@gmail.com

Yiwei Yang
University of California, Santa Cruz
Santa Cruz, United States
yyang363@ucsc.edu

Minghui Jiang
Alibaba Group
Hangzhou, China
shimei.jmh@alibaba-inc.com

Xiangyu Gao
University of Washington
Seattle, United States
xiangyug@cs.washington.edu

Jianchang Su
University of Connecticut
Storrs, United States
jianchang.su@uconn.edu

Wenan Mao
Alibaba Group
Hangzhou, China
wenan.mwa@alibaba-inc.com

Wei Zhang
University of Connecticut
Storrs, United States
wei.13.zhang@uconn.edu

Dan Williams
Virginia Tech
Blacksburg, United States
djwillia@vt.edu

Andi Quinn
University of California, Santa Cruz
Santa Cruz, United States
aquinn1@ucsc.edu

Abstract

GPU resource management policies govern how a system places memory and schedules GPU tasks. They play an important role in determining application performance, and the ideal policy for a given deployment often depends on low-level details such as the application, workloads, and hardware. AI agents would make it possible to automatically optimize GPU resource management policies, but require three properties of a system: safety, to ensure that a broken policy cannot cause a system crash; dynamism, to enable policy changes without restarts; and full-stack, to ensure that the policy can observe device and application states and control low-level hardware. Unfortunately, existing techniques for GPU resource management fail to meet these needs.

We introduce gpu_ext, a new system that provides an OS interface for GPU resource management that is safe, dynamic, and full-stack by turning to eBPF: gpu_ext policies include eBPF callback functions that the system invokes on a specified event (e.g., when code reaches a function). The system includes a novel execution model that separates the effect of a callback from its triggering event. Namely, gpu_ext supports policies that asynchronously transition resources between states, potentially invoking multiple transitions at a single point-in-time. We implemented gpu_ext on the Linux NVIDIA GPU driver and find that AI-generated policies improve throughput by 1.76× over application-tuned baselines and reduce tail latency by up to 2×, without application modifications. Reimplementations of seven prior policies with gpu_ext achieve performance close to their native implementations. Moreover, we find that there were zero OS kernel panics through the optimization process, highlighting gpu_ext's safety.

CCS Concepts: • **Software and its engineering → Virtual memory**; **Scheduling**; *Formal software verification*; • **Computer systems organization → Graphics processors**.

Keywords: GPU resource management, eBPF, OS kernel extensibility, memory management, scheduling, unified virtual memory

*Both authors contributed equally to this research.



This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '27, Heraklion, Greece
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3012-2/2027/04
https://doi.org/10.1145/3845814.3850045

ACM Reference Format:
Yusheng Zheng, Tong Yu, Yiwei Yang, Minghui Jiang, Xiangyu Gao, Jianchang Su, Wenan Mao, Wei Zhang, Dan Williams, and Andi Quinn. 2027. Safe and Programmable OS-Level GPU Resource Management with eBPF. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3845814.3850045

1 Introduction

GPU resource management policies critically influence application performance. These policies include memory placement, migration, and GPU task scheduling. Recent research shows their substantial impact on Unified Virtual Memory (UVM) placement [4, 28, 34, 37] and GPU scheduling strategies [13, 49, 58]. For instance, optimal eviction and prefetch policies can improve performance by up to 73% under memory oversubscription [14, 28, 33].

Unfortunately, finding an optimal GPU resource management policy is challenging. The optimal memory and scheduling policies vary significantly across workloads, hardware, and deployment contexts [1, 5, 32, 35, 44, 47, 50, 57], with no one policy dominating the others. For example, aggressive memory prefetching improves latency for LLM MoE inference by reducing memory stalls [60, 63], but increases the latency of graph analytics queries because it over-saturates memory bandwidth [14]. Additionally, throughput-oriented scheduling improves average latency of vector search but degrades tail latency for interactive inference [1, 13].

We argue that the community should employ AI agents to automatically optimize programmable GPU resource management policies. AI-driven exploration can handle the vast, workload-specific policy space that makes manual tuning challenging. Agents can now generate complex code and optimize policies in domains such as compiler heuristics [2], datacenter scheduling, LLM serving [8, 20], training strategies [11], and CPU scheduling [66]. In these use-cases, agents are generating fully fledged policy programs rather than simply tuning configuration parameters.

We derive three properties that systems should provide to enable AI agents to optimize programmable GPU resource management policies effectively without sacrificing system health. Namely, a system should support programmable policy interfaces that are safe, dynamic, and full-stack. A policy interface is safe if it ensures that new policies cannot cause the system to crash; dynamic if it supports policy updates without requiring the system and applications to be restarted; and full-stack if it supports cross-application and device-internal visibility, as well as low-level hardware control.

Existing programmable GPU resource management techniques fail to simultaneously meet these requirements. User-space frameworks [7, 17, 38] are safe and dynamic but are not full-stack since policies lack cross-application visibility and hardware control. Other approaches modify the OS kernel driver [13, 30, 31, 58, 65], which fails to meet all three of the requirements.

In this paper, we propose gpu_ext, a new system that provides an OS interface for GPU resource management policy that is safe, dynamic, and full-stack. Similar to prior work on resource management [22, 54, 68], gpu_ext supports eBPF-based policies. Users write callback functions that manage system resources and are called on user-defined events (e.g., when the system reaches a specific code location, or periodically). gpu_ext's use of eBPF ensures safety, since the system uses an eBPF-style verifier before running new policies, and dynamism, since it can replace policies without service interruptions. To provide a full-stack interface, gpu_ext supports eBPF callbacks that execute within the OS kernel driver and GPU device.

gpu_ext models GPU resource management as transitions on resource state machines (§3.2). State machines define valid states and transitions: GPU memory chunks transition between host memory and device memory through eviction and prefetching; compute contexts transition through states such as running, ready, and finished via scheduling and preemption operations. eBPF policies specify the transitions that the system should take on its GPU resources.

Alas, applying the CPU eBPF programming model to GPU resource management is not straightforward. The GPU is a discrete processor with its own execution model and resources, and policy decisions made in the host take effect on a separate device. In particular, we identify three challenges: first, cross-device operations like migrating data or dispatching commands take milliseconds, too slow for synchronous callbacks and too costly for purely reactive decisions. Second, unlike independent subsystems on the CPU, limited device memory and bandwidth further couple memory and scheduling, as decisions about data placement directly affect which workloads can execute. Third, GPU applications employ OS kernel bypass, which requires gpu_ext to support eBPF callbacks from events in user-space applications. Traditional eBPF does not allow such callbacks to modify OS state required to implement resource management policies because of safety concerns.

gpu_ext addresses these challenges through a novel execution model for eBPF-based resource management that separates the effect of a callback from its triggering event. Prior eBPF-based systems, such as sched_ext [54], cache_ext [68], and XDP [22], primarily execute policy logic at subsystem-defined hooks, with decisions applied within their respective subsystems. In contrast, gpu_ext decouples policy execution from resource state transitions, enabling callbacks at user-defined events to initiate multiple asynchronous operations across the CPU–GPU boundary. In particular, gpu_ext callbacks invoke transitions when a given event occurs, and the system asynchronously transitions the resource between states. gpu_ext callbacks can initiate multiple state transitions within a single event. Finally, gpu_ext supports user-defined events that can execute policy code from anywhere in the GPU stack (application, OS kernel driver, or GPU device). Critically, to enable safety with user-defined events, gpu_ext policies only initiate predefined transitions; they cannot create invalid transitions that could corrupt resources.

We implement gpu_ext on Linux by extending the NVIDIA open GPU OS kernel modules [39] and providing an eBPF

SIMT verifier and JIT runtime for GPU devices. We evaluate gpu_ext across LLM inference, GNN training, vector search, and multi-tenant scenarios. Policies based on gpu_ext improve throughput by up to 4.8× over framework offloading and 1.76× over application-tuned baselines, and reduce tail latency by up to 2×, with low overhead and no application source-code changes or restarts. Across seven prior policies, the main metrics in Figure 15 differ by less than 2% between gpu_ext and native implementations. We show that gpu_ext enforces safety by using AI agents to generate all of the 59 evaluated eBPF policies, including eviction ordering, prefetch strategies, scheduling logic, and device-side work-stealing. gpu_ext ensured that there were zero OS kernel panics throughout this process. In summary, our contributions are:

- an execution model for eBPF-based resource management that separates the effect of a callback from its triggering event.
- gpu_ext, a system that provides a programmable GPU resource management policy interface that is safe, dynamic, and full-stack.
- an evaluation across four GPU workloads, where gpu_ext policies improve throughput by up to 4.8× and reduce tail latency by up to 2×, differ in performance from prior work policies by less than 2%, with all policies generated by AI agents and no OS kernel panics.

2 Background and Motivation

2.1 GPU Systems and Workload Diversity

As shown in Figure 1, we focus on three GPU resource management domains across the software stack: memory placement and migration, compute scheduling, and on-device thread scheduling. User-space frameworks leverage application-level information to manage GPU resources: vLLM [32] optimizes LLM request scheduling and KV-cache management, Salus [64] enables fine-grained GPU sharing among deep learning jobs, and PILOT [48] manages unified memory footprints across GPU tenants. Kernel drivers manage memory, scheduling, and multi-tenant isolation via privileged hardware mechanisms (MMU, interrupts), typically using fixed, application-agnostic algorithms (e.g., LRU eviction) under Unified Virtual Memory (UVM), which transparently handles oversubscription through host-device page migrations. GPU devices execute kernels via SIMT (Single Instruction, Multiple Threads); on NVIDIA hardware, 32-thread warps run instructions in lockstep on SMs, with internal states (e.g., warp access patterns) opaque to host drivers.

GPU workloads exhibit diverse resource access patterns. Results from gpu_ext's eBPF-based tracing tools (§5) confirm this diversity: Figure 2 captures memory page-fault addresses over time under UVM, showing that LLM MoE

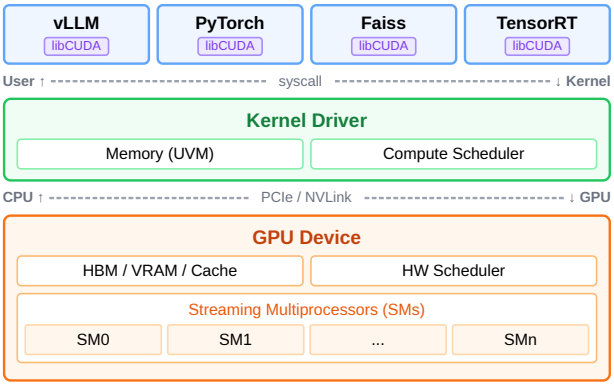


Figure 1. GPU system stack: user-space frameworks, kernel driver (memory and scheduling), and GPU hardware.

inference exhibits periodic stride bursts from expert activations; GNN training shows irregular random access from pointer-chasing graph traversal; and vector search alternates between sequential scans (index build) and random probes (query) [27, 44]. Figure 3 reveals a 127× SM scheduling load imbalance in a vector-add GPU kernel, invisible to host-side drivers.

These diverse patterns require workload-specific policies across all three domains. MoE inference benefits from workload-aware expert caching and offloading [24, 60, 63], while GPU graph workloads exhibit irregular memory access patterns that motivate different memory management strategies [35, 57].

Compute scheduling requires adaptive priority and preemption policies as multi-tenant demands fluctuate [13, 58], and SM load imbalance benefits from custom thread scheduling. Yet current drivers and firmware enforce a single fixed policy.

2.2 Agent-Based Policy Exploration

Agentic OS policy exploration fits the large GPU policy space but requires programmable interfaces missing in current GPU stacks. Unlike parameter-tuning methods, AI agents [12, 66] directly generate and iteratively refine policy logic to explore richer, workload-specific strategies. However, existing GPU stacks provide only fixed knobs (e.g., `cudaMemAdvise`, driver module parameters), lacking programmability needed for complex heuristics such as workload-aware eviction or adaptive prefetching.

Effective GPU policy optimization requires an OS interface with three key properties: (1) *safe*, robustly handling generated policies without kernel panics or GPU corruption; (2) *dynamic*, rapidly deploying and iterating policies at runtime without application restarts or kernel rebuilds; and (3) *full-stack*, observing GPU-internal states, coordinating across tenants, and controlling driver-level decisions like eviction and fault-path prefetching.

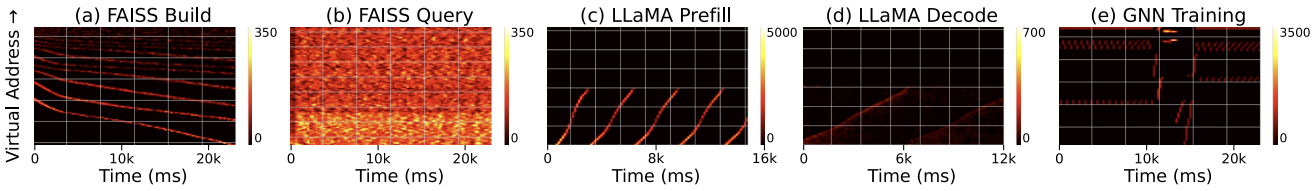


Figure 2. Page-fault address traces over time across four workloads.

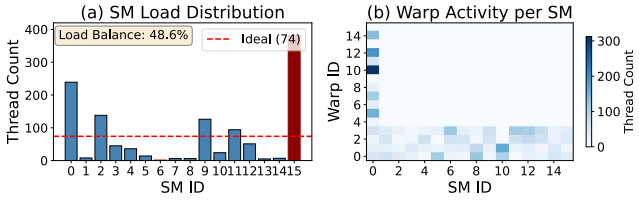


Figure 3. GPU thread scheduling imbalance observed via eBPF tracing. (a) SM load distribution: 127× imbalance. (b) Warp activity heatmap across SMs.

2.3 Limitations of Existing Approaches

Existing approaches fail to simultaneously provide a safe, dynamic, and full-stack programmable OS interface for GPU resource management.

Driver-Level Policies. Driver-level approaches either expose fixed knobs (e.g., `cudaMemAdvise`, driver module parameters) that select among preset behaviors, or embed custom logic directly into driver code. Systems like TimeGraph [30], Gdev [31], and OS kernel interception methods [65] embed scheduling and memory-management logic into driver modules. More recent work such as GPREEMPT [13] and GCAPS [58] add priority-based preemption, and LithOS [10] re-implements runtime stacks with OS-like abstractions. Despite improved performance, all these approaches require OS kernel or driver modifications to change policy: they are neither safe (bugs can crash the OS kernel) nor dynamic (policy changes require recompilation and restarts), and still lack GPU-internal observability for full-stack control.

Host User-Space Runtimes and Libraries. User-space frameworks [7, 10, 17, 38, 49, 51] offer programmability but face three limitations. First, policies are application-bound, requiring per-framework implementation and preventing reuse across ecosystems. Second, process-level isolation prevents cross-tenant visibility and coordinated memory-scheduling decisions across tenants [59, 64]; hardware partitioning such as MIG [42] imposes fixed resource boundaries that cannot be dynamically rebalanced. Third, user-space code cannot access privileged driver mechanisms, leaving policies constrained to GPU kernel launch boundaries and unable to execute on latency-critical paths (e.g., page-fault handlers), perform precise preemption on driver or device states, or exploit GPU-side warp-level access patterns. In summary, these frameworks are safe and dynamic but not full-stack.

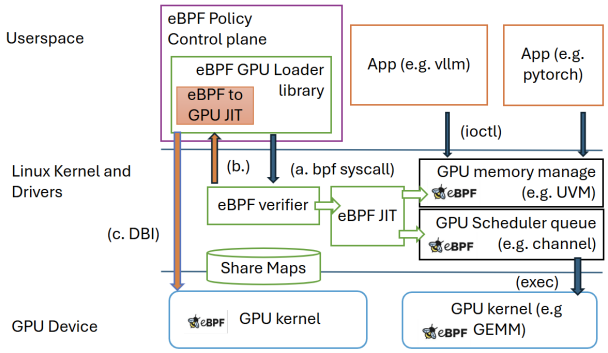


Figure 4. gpu_ext architecture. (a) eBPF syscall deploys policies to verifier and driver hooks; (b) GPU JIT after verification; (c) DBI injects trampolines into GPU kernels.

3 gpu_ext Design

This section presents the design of gpu_ext, a programmable OS subsystem for safe, dynamic, full-stack GPU resource management. We describe the system workflow (§3.1), resource state machines (§3.2), GPU-specific eBPF challenges (§3.3), and our asynchronous execution model (§3.4).

3.1 gpu_ext Workflow

gpu_ext extends eBPF (extended Berkeley Packet Filter) to GPU resource management. eBPF programs are callbacks attached to hooks such as `struct_ops`, `kprobe`, and `uprobe`, enabling safe execution within the OS kernel via load-time verification and JIT compilation. The verifier ensures termination, memory safety, type correctness, and restricts calls to approved helpers and kfuncs (OS kernel functions). As shown in Figure 4, gpu_ext extends this pipeline to GPUs: policies build and load with standard eBPF tools, contain both host and device components, run across the host driver and GPU, are verified with SIMT-aware checks (§3.5), JIT-compiled to CPU or GPU code (e.g., PTX), use cross-layer maps, and are dynamically installed via OS kernel hooks and DBI without application or driver restarts.

3.2 Resource State Machines

gpu_ext defines three resource domains. Memory chunks move between host and device memory via eviction and prefetching. GPU kernels (launched tasks) and GPU thread blocks (parallel execution units within kernels) transition among ready, running, and finished states through scheduling and preemption operations. Extending the model to storage or CXL tiers would require corresponding placement states and backend-validated transfers.

3.3 GPU-Specific Challenges for eBPF Extensibility

CPU eBPF does not directly transfer to GPUs. CPU eBPF callbacks make synchronous decisions at fixed transition points, immediately affecting local resources, and subsystems are extended independently. In contrast, GPUs are discrete processors: host policy decisions affect resources asynchronously on a separate device. Thus, three architectural mismatches introduce key challenges:

Cross-device latency. CPU eBPF callbacks execute synchronously, but GPU operations (e.g., page migration across PCIe) take milliseconds (§2.1). GPU policies must proactively initiate asynchronous actions instead of reacting only at faults.

Memory-scheduling coupling. CPU extensibility treats scheduling and memory independently, but GPU page faults simultaneously trigger memory events and scheduling stalls due to limited device memory. GPU policies must jointly coordinate these domains.

OS kernel bypass. GPU runtimes bypass the OS kernel (e.g., user-space command buffers, doorbells), revealing only coarse-grained resource events to drivers, not fine-grained GPU kernel launches or application phases. GPU policies must attach to user-space application functions, yet standard eBPF prohibits these callbacks from modifying OS state required for resource management. Exposing GPU driver controls without addressing these mismatches risks resource leaks, hardware hangs, or OS kernel panics.

3.4 Asynchronous Execution Model

To address these challenges, gpu_ext separates callback triggers from their effects (Figure 5). Policies, as verified eBPF programs invoking OS kernel functions (*kfuncs*), can attach to driver events, user-space uprobes, and device-side events. Each trigger asynchronously initiates multiple proactive or deferred state transitions across memory and scheduling domains without blocking, coordinated through eBPF workqueues. For example, a uprobe on `cudaLaunchKernel` concurrently initiates a 6.6 ms DMA prefetch based on device observations and preempts another tenant to reduce resource contention.

To ensure OS kernel safety, verified kfuncs restrict policy actions to two dimensions: *selection* (choosing resources, e.g., eviction order, tenant preemption priority) and *timing* (when

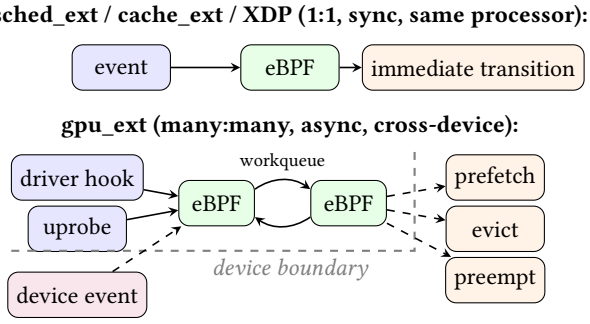


Figure 5. Policy execution model: CPU eBPF policies vs gpu_ext.

to act, e.g., proactive prefetch). The driver always controls underlying state machines, so policies only select ordering and timing of driver-managed operations, cannot allocate memory or remove tasks, and fall back to the driver's original behavior under resource pressure or misconfiguration. Currently, a single system-wide policy can differentiate tenant priorities, similar to sched_ext. Independent tenant policies are future work and would require separate permissions, map isolation, and execution budgets.

3.5 Runtime Verification and Optimizations

gpu_ext's threat model mirrors standard eBPF: administrators are trusted, but policy code may be buggy or misconfigured. Before a policy runs, the verifier checks its memory accesses and termination. When a policy requests a resource change, the driver checks whether the change is valid for the resource's current state. Table 1 summarizes the rules and their enforcement.

Trusted computing base. The trusted computing base (TCB) comprises the host OS, GPU driver and firmware, policy verifiers and compilation toolchain, and gpu_ext's driver module, hooks, helpers, and transition validators. Policy safety assumes correct operation of the GPU hardware and this TCB.

3.5.1 Transition Validation. Before applying a request, the driver checks resource ownership, whether the observed state is still current, and whether the requested transition is allowed. The driver must prevent other operations from invalidating these checks before it applies the request. For mutable resources such as eviction queues, the driver synchronizes validation and the resulting state change as one step. Long-running transfers remain asynchronous under the native driver's control.

Within one callback, repeating an identical request has the same effect as issuing it once. If one callback requests incompatible actions on the same resource, the driver rejects those requests. The driver leaves the eviction order unchanged

Illegal behavior	Safety property	Handling
Out-of-bounds access or unbounded loop	Memory bounds and termination	Reject program
Lane-dependent branch	Uniform control flow	Reject program
Varying shared-map value or atomic target	Uniform updates to shared state	Reject program
GPU-wide barrier	Bounded synchronization	Reject program
Stale eviction request	Current resource state	Preserve state
Conflicting requests	Request consistency	Reject requests
Invalid prefetch range	Valid migration range	Native prefetch

Table 1. Safety rules and representative rejected behaviors. Program rejection occurs before execution. Resource requests are checked before transition initiation.

when it rejects a reordering request and uses native prefetch behavior for invalid prefetch actions or ranges.

3.5.2 SIMT-aware Verification. Because GPUs execute in SIMT (Single Instruction, Multiple Threads) mode, device-side eBPF programs must maintain warp-level execution semantics within eBPF's scalar model. gpu_ext enforces warp-uniformity via SIMT-aware verification: it distinguishes *warp-uniform* (identical across warp threads, e.g., region IDs) from *lane-varying* (thread-local state) values. After checking memory bounds and termination, the verifier tracks how values depend on their inputs: constants are uniform, while values derived from lane identifiers may differ across lanes. This analysis follows computations and control flow, including loops.

When control-flow paths merge, the verifier treats a value as lane-varying if it may differ across lanes on any incoming path. Lane-varying values cannot affect control flow or external side effects. Branch conditions, loop bounds, and map-update keys must be warp-uniform or warp-aggregated. Shared-map values and atomic target addresses must also be uniform so that lanes update shared state consistently. The verifier rejects programs when it cannot establish the required uniformity. For example, a branch on lane ID or a shared-map update derived from it is rejected. Additionally, gpu_ext forbids GPU-wide barriers, global synchronization, non-uniform atomics, and it bounds resource usage per policy hook.

3.5.3 Warp-level Execution Optimizations. Even after SIMT verification, scalar per-thread eBPF execution incurs overhead (e.g., duplicate map updates). To resolve this, gpu_ext introduces warp-level aggregated execution: lanes compute local results without affecting control flow or shared state; these results are aggregated warp-wide, and a warp leader executes the policy once, broadcasting decisions back (§4).

3.5.4 Hierarchical Cross-layer Maps for State Coordination. To share state across CPU and GPU contexts, gpu_ext introduces cross-layer maps: logically unified key-value stores accessible by host driver hooks, GPU-side policies, and user-space control planes, verified through standard eBPF. Physically, maps are hierarchical shards dynamically placed by the runtime based on access patterns, latency, and capacity constraints (§4). Following eBPF's philosophy, gpu_ext maps use eventual consistency: periodically merging device-local shards into canonical snapshots at GPU kernel completions. This model suffices as policies rely on approximate statistics; occasional staleness may affect optimality but cannot violate correctness invariants like memory integrity. When access patterns change faster than the synchronization interval, policies can synchronize more frequently, trading overhead for freshness. When application heuristics interfere with a policy, the policy can adjust its prefetch aggressiveness using observed access patterns and memory pressure. Maps can hold individual records for host or GPU access and counters that merge across shards through associative operations such as addition.

3.6 Example: MoE Expert Offloading

We illustrate this model using Mixture-of-Experts (MoE) offloading, where models exceed GPU memory capacity. Since all experts do not fit and activate concurrently, the GPU driver continuously evicts and migrates pages. The default driver's LRU eviction and prefetching are unaware of expert activations, causing costly PCIe DMA stalls.

A gpu_ext policy uses three cooperating eBPF programs at different layers (Figure 6), coordinating via a shared cross-layer map. First, during GPU kernel execution, a device-side eBPF handler inserted at warp entry tracks warp-level accesses to expert pages, recording per-page access counts for each warp in a hierarchical cross-layer map. Second, a uprobe eBPF program attached to `cudaLaunchKernel` proactively reads these counts to predict upcoming expert activations and asynchronously initiates cross-device prefetches (`bpf_gpu_prefetch_async`), overlapping the millisecond-scale memory migration with computation to avoid expensive stalls (*timing*). If memory pressure arises, this same program coordinates across memory and scheduling by preempting lower-priority GPU tasks via `gdrv_sched_preempt` (*selection*). Finally, upon memory eviction (*evict_prepare*), a driver-side eBPF handler leverages the same expert access counts to evict cold pages first (LFU ordering), protecting hot experts (*selection*). Unlike framework-level offloading that often migrates experts as atomic units, gpu_ext operates at page granularity, enabling finer-grained compute-transfer

```
SEC("gpu_dev/warp_entry")
int moe_observe(gdev_page_access_ctx_t *ctx){
  u64 va = ctx->va & EXPERT_MASK;
  u64 *freq = bpf_map_lookup_elem(
                &expert_freq, &va);
  if (freq) fetch_and_add(freq, 1);
  return 0;
}
SEC("uprobe/cudaLaunchKernel")
int moe_prefetch(struct pt_regs *ctx) {
  u64 next = predict_next_expert(ctx);
  bpf_gpu_prefetch_async(
                next, EXPERT_SIZE);
  if (bpf_gpu_mem_pressure() > THRESHOLD)
    gdrv_sched_preempt(BE_CONTEXT);
  return 0;
}
SEC("gpu_mem/evict_prepare")
int moe_evict(gdrv_mem_remove_ctx_t *ctx){
  gmem_region_t *r = ctx->region;
  u64 *freq = bpf_map_lookup_elem(
                &expert_freq, &r->va);
  if (freq && *freq > HOT_THRESHOLD)
    bpf_gpu_move_tail(&ctx->evict_list, r);
  else
    bpf_gpu_move_head(&ctx->evict_list, r);
  return 0;
}
```

Figure 6. MoE expert offloading policy

overlap. Each eBPF program captures partial state at different layers; cross-layer maps unify this fragmented visibility to orchestrate state transitions across host and GPU.

4 Implementation

We implemented gpu_ext on Linux 6.15, extending the OS kernel with a standalone file (~1 KLOC) that handles hook registration and verifier extensions. We integrated lightweight instrumentation (~100 LOC) into NVIDIA's open GPU OS kernel modules to expose memory and scheduling state transitions as GPU-specific kfuncs (§4.1). Our user-space loader and JIT infrastructure (~10 KLOC) builds upon bpftime's framework [67] for device-side code generation, while an LLVM-based backend (~1 KLOC) translates the restricted gpu_ext eBPF instruction subset into GPU device code (PTX) (§4.2). The host design aligns with Linux HMM/`migrate_vma` [53] and DRM scheduling [52], and device code generation could target vendor-neutral IR [9] with their LLVM backend. Porting beyond NVIDIA would require adapting driver hooks, instrumentation, and code generation. Future hardware or firmware support for validated transition hooks and runtime statistics could simplify such ports.

4.1 Host-side Integration

Memory State-Transition Hooks. gpu_ext extends the NVIDIA Open GPU OS Kernel Modules [39], instrumenting the UVM module at four code locations: *activate* (chunk allocation), *access* (page fault), *evict_prepare* (eviction candidate selection), and *prefetch* (proactive migration). At each hook, the driver constructs a context (e.g., 2MB-aligned region range, fault address, block size) and invokes the attached eBPF handler. Eviction hooks operate at 2MB granularity; prefetch hooks operate at page granularity and can prefetch as many pages as needed. We extend eBPF to allow each handler to initiate asynchronous effects via kfuncs such as `bpf_gpu_prefetch_async` for cross-device DMA or `gdrv_sched_preempt` for scheduling preemption with `bpf_wq` (workqueue on background threads). The driver maintains a doubly-linked eviction list; handlers may reorder regions via provided helpers, but the driver retains eviction authority under memory pressure.

Scheduling Hooks. We instrument Time-Slice Group (TSG) code locations, including initialization (`task_init`) and teardown (`task_destroy`). The eBPF policy can configure scheduling parameters such as TSG priority, timeslice duration, and runlist interleave frequency via the `bpf_gpu_set_attr` kfunc. Policies triggered from uprobes or tracepoints can also preempt running GPU kernels precisely with preemption-request kfuncs.

4.2 GPU Runtime

Verification and JIT Compilation. The prototype uses Linux's eBPF verifier for kernel-resident host programs. For device-side programs, it uses PREVAIL [16, 46] with GPU helper and map models, followed by the uniformity analysis and SIMT checks in §3.5. After verification, the user-space backend translates device-side bytecode into PTX, inlining helper functions and map accesses. Unlike the workshop version [62], which relies on atomic synchronization, gpu_ext avoids cross-SM primitives to prevent hardware stalls.

Device-Side State-Transition Hooks. gpu_ext hooks device-side state transitions (GPU kernel launch, memory access, execution-phase boundaries) by using ptrace to pause applications, intercepting CUDA runtime APIs, extracting GPU kernel PTX, inserting binary trampolines, and reloading instrumented GPU kernels without application recompilation or restart. Applications can also load the runtime at startup using `LD_PRELOAD`. For applications without PTX, host-side policies still operate through driver hooks, and an alternative instrumentation inserts eBPF code into SASS. Trampolines insert prologue/epilogue logic for SIMT register preservation. In each trampoline, per-lane inputs are aggregated using `__ballot_sync` and `__shfl_sync`, with a warp leader selected via `__ffs(__activemask())` to execute the scalar eBPF handler once per warp, broadcasting

results via shuffle instructions. For block-level scheduling, gpu_ext implements policies using persistent GPU kernels that poll and claim logical work units; on Blackwell GPUs, we use cluster launch control APIs [41] to schedule thread blocks. For memory observation, gpu_ext attaches eBPF handlers at memory instructions, GPU kernel function boundaries, and tracepoints to capture access patterns and issue policy hints to the host driver via cross-layer maps (e.g., device L2 prefetch can trigger a host callback for extended prefetch).

Hierarchical Maps and Cross-Domain State. We realize cross-layer maps (§3.5.4) by placing shards in GPU-accessible memory tiers: cold state in host DRAM, hot state in GPU global memory, and per-warp data in GPU shared memory. A runtime daemon asynchronously flushes GPU-local shards to host-visible canonical instances, providing periodically refreshed snapshots without synchronization overhead. Shard updates occur at warp granularity through warp-leader threads without GPU atomic operations: handlers accumulate counters in warp-local registers, aggregate per warp, and store into GPU-local shards for periodic host flush.

5 Evaluation

Our evaluation addresses two research questions:

- **RQ1 (Policy Expressibility and Benefits):** Can AI agents safely express GPU resource-management policies, including those from published state-of-the-art systems, through gpu_ext? Do these policies improve performance?
- **RQ2 (Mechanism Cost):** What runtime overhead does gpu_ext add for hooks, observability, and policy execution?

5.1 Methodology and Setup

We evaluate gpu_ext on two machines: Server A with Intel Core Ultra 9 285K (24 cores), 128 GB DDR5 RAM, and NVIDIA RTX 5090 GPU; Server B with dual Intel Gold 6138 (80 cores), 256 GB RAM, and NVIDIA P40 GPU. Unless otherwise specified, experiments run on Server A, averaging 10 trials using the geometric mean. We compare gpu_ext policies against three classes of baselines: (1) driver defaults (UVM LRU/FIFO eviction, native GPU scheduler), (2) application-level hints (`cudaMemAdvise`), and (3) framework-managed policies (e.g., llama.cpp `ncmoe`, vLLM CPU offload, LMCache) where applicable. We use PyTorch [45] (version 2.9.0) and vLLM [32] (version 0.11.0) with custom allocators, llama.cpp [15] (version 7101), and Faiss [27] (version 1.13.0) with UVM build config. All tests with gpu_ext policies require no application source modifications.

5.2 RQ1: Policy Expressibility and Benefits

5.2.1 Single-Tenant Memory and Scheduling Policies.

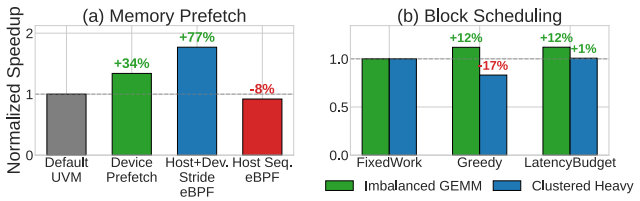


Figure 7. Single-tenant policy evaluation (normalized speedup, higher is better): (a) memory prefetch policies on vector-add; (b) block scheduling policies on GEMM.

Memory Prefetch Policies. We evaluate host-device prefetch coordination using a vector-add GPU kernel [40] with a strided access pattern and a 40 GB working set (1.25× oversubscription). Device-side L2 prefetch instructions (`prefetch.global.L2`) trigger non-blocking page faults, while host-side callbacks extend prefetching to additional pages. As shown in Figure 7a, device-only prefetch (45 LOC) at thread entry achieves 1.34× speedup while combined host-device stride-based eBPF prefetch (472 LOC host + 45 LOC device) achieves 1.77×. However, sequential prefetch (375 LOC) degrades performance by 8% due to pattern mismatch.

Single-Tenant Block Scheduler. We apply a gpu_ext-based thread block scheduler to workloads with load imbalance. Using cluster-style launch where persistent worker blocks pull work units via `should_try_steal`, we evaluate three policies: FixedWork (static block assignment, no scheduler), Greedy (always-steal, 16 LOC), and LatencyBudget (workload-aware with per-block time limits, 19 LOC). Figure 7b compares these policies across two GEMM workloads. Under moderate imbalance, both Greedy and LatencyBudget reduce latency by ~11%. Under heavy-tail workloads where 10% of blocks perform 100–200× more work, Greedy increases latency by 20% due to contention on heavy blocks. LatencyBudget, which caps per-block stealing time, matches the fixed-work baseline. This highlights gpu_ext's programmability: different workloads require different scheduling strategies.

5.2.2 Agent-Driven Policy Exploration Case Studies.

We provided an AI coding agent (Claude Code with Opus 4.5) with gpu_ext's eBPF interfaces; the agent autonomously generated all policy logic without human review or guidance, while humans authored only the initial framework and benchmark harness. All 59 policies evaluated in the agent exploration study were produced via a fully automated generate-test-iterate loop. Over 20 days, the agent performed 974 benchmark runs and 244 code edits, exploring 59 distinct policies and consuming 5.5M tokens ($388 API cost); approximately 75% of explored policies yielded negative results before being iteratively corrected. We documented 50 safety events (24 logic bugs, 18 performance regressions, 2 verifier

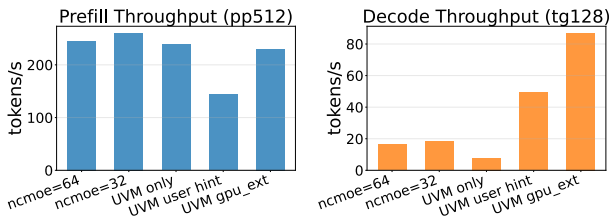


Figure 8. Prefill and decode throughput for GPT-OSS-120B MoE on llama.cpp.

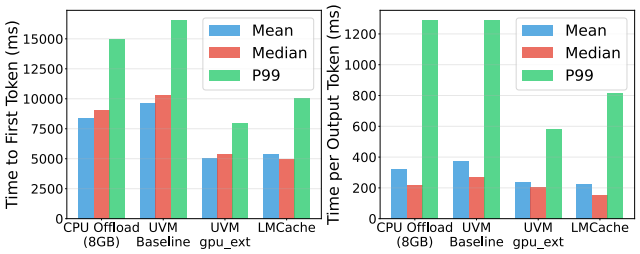


Figure 9. Time-to-first-token and time per output token on Qwen-30B MoE inference with vLLM.

rejections, 2 GPU verifier-caught overflows, 4 others), all recovered without OS panics or data corruption, typically within minutes and without application restarts. Equivalent bugs in modified GPU drivers risk kernel panics, requiring recompilation and service interruptions rather than instant policy detachment.

Expert Offloading (llama.cpp GPT-OSS-120B). We evaluate gpu_ext on GPT-OSS-120B [43] MXFP4 MoE (59 GB) in llama.cpp on RTX 5090 (32 GB), requiring 1.84× oversubscription. We compare five configurations: framework-managed CPU offloading (`ncmoe=32/64`), default UVM, UVM with application-level hints (`cudaMemAdvise`), and a gpu_ext host-device prefetch policy (§3.2). For decode (memory-bound), the expert-offloading policy in gpu_ext achieves **1.76× speedup** over application-tuned UVM hints (`cudaMemAdvise`) and 4.8× over framework offloading (`ncmoe=64`) (Figure 8). These hints improve decode over default UVM but require application modification and degrade prefill by 40%; the gpu_ext policy further improves decode throughput while preserving prefill throughput within 4% of default UVM without application source-code changes. For prefill (compute-bound), framework offloading (`ncmoe=32`) achieves 13% higher prefill throughput than this policy, but decode dominates end-to-end latency, so the decode improvement outweighs the prefill gap.

The agent arrived at the final policy through iterative exploration: sequential prefetch degraded decode throughput due to mismatch with MoE stride patterns (consistent with the 8% degradation in Figure 7a), leading to stride-based prefetch aligned with regular weight layout within each expert layer. Device-side eBPF tracing of per-warp memory accesses revealed that expert activations follow predictable stride sequences and identified frequently-accessed shared layers. For eviction, the agent tested FIFO ordering before converging on LFU, which retains frequently-accessed shared layers while evicting cold expert pages. The final policy combines device-side access observation (45 LOC), driver-side stride prefetch (472 LOC), and driver-side LFU eviction (304 LOC), totaling ~820 LOC (§3.2).

KV-cache Offloading (vLLM Qwen-30B MoE). We evaluate gpu_ext on the Qwen-30B [61] FP8 MoE model (≈30GB) on RTX 5090 (32GB). The model nearly fills GPU VRAM, so the default no-offload configuration OOMs under load. We stress KV-cache growth by sending 100 concurrent requests from ShareGPT [3] (single-round, no prefix caching), resulting in 36–40GB total footprint (≈60K tokens KV-cache). Existing solutions offload KV-cache or experts but not both; UVM attempts on-demand migration of both, leading to thrashing. We compare vLLM's CPU-offload mode (`--cpu-offload-gb 8`) against a UVM policy in gpu_ext. This policy improves mean and p99 time-to-first-token by 1.7–2× and reduces mean time per output token by 27.3% relative to vLLM's default framework-managed offload (Figure 9), achieving similar mean latencies to LMCache [36], a state-of-the-art KV-cache offloading framework. Notably, UVM without gpu_ext policies performs worse than vLLM's offload, while this policy performs similarly to the best framework offloading, with better tail latency, dynamically and transparently.

Under memory pressure, the main challenge is managing two competing data types: KV-cache, exhibiting temporal locality (recent tokens accessed frequently during attention) and per-request spatial locality, and model weights with stride patterns in matrix operations. A uniform sequential prefetch initially caused mutual thrashing between KV-cache and weight pages. Using device-side tracing, the agent explored region-differentiated prefetch strategies, converging on adaptive aggressiveness guided by PCIe utilization and region type (stride for weights, sequential for KV-cache, 375 LOC), coupled with LFU eviction (304 LOC) retaining frequently accessed pages. The resulting policy (totaling ~680 LOC) effectively prevents thrashing seen with default LRU.

We also evaluate read/write scheduling for KV-cache offloading using LMCache's local-disk backend. An agent-developed policy prioritizes reads over background writes while limiting each write's total deferral to 200 ms. Across five paired runs, median read p99, measured from scheduled request arrival, falls from 323.707 ms with FIFO to 123.141 ms with the native policy and 118.097 ms with the same policy

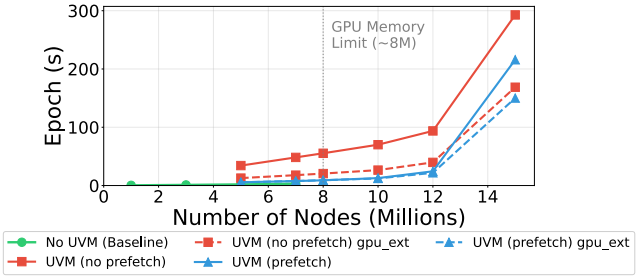


Figure 10. GCN training epoch time vs graph size on PyTorch.

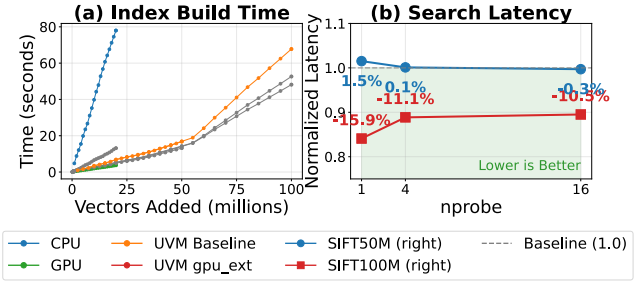


Figure 11. Faiss index build time and search latency normalized to default UVM under oversubscription.

in gpu_ext. The median paired reduction with the gpu_ext policy relative to FIFO is 61.9%.

Graph Neural Networks (GNN Training). We evaluate gpu_ext on PyTorch-based GCN training with random graphs of varying sizes (1M–15M nodes, 10 edges/node). Figure 10 shows epoch time across five configurations: native GPU allocation (no UVM), UVM with and without user-space prefetching (`cudaMemPrefetchAsync`), and each UVM configuration with the prefetch policy in gpu_ext. With UVM enabled via a custom allocator, each epoch allocates memory on CPU and migrates to GPU on demand, incurring overhead. User-space frameworks can pre-migrate pages via `cudaMemPrefetchAsync`, achieving 5.5× speedup at moderate oversubscription, but this requires application modification. Without user hints, this policy optimizes page fault handling via aggressive prefetching, achieving 2.65× speedup transparently. Combined with user-expressed prefetching, this eBPF policy achieves 1.44× additional speedup by prefetching larger regions and reducing blocking faults. Native GPU allocation (no UVM) achieves the highest performance when data fits in memory (1.89 s at 5M nodes) but fails with OOM beyond ~8M nodes. UVM with this prefetch policy enables training at 15M nodes (2.17× oversubscription) with lower epoch times than default UVM, without application modification.

The policy uses sequential prefetch (375 LOC) with a uprobe tracing PyTorch allocations, determining prefetch ranges via device-traced history. The agent tested deeper lookahead after single-block-ahead prefetch (K=1) improved epoch time by 21%, but found K=2 and K=3 degraded by 16%, and adaptive K (up to 6) by 46%. Suspecting lock contention initially, the agent identified PCIe bandwidth saturation as the true bottleneck, confirming one-block lookahead as optimal.

Faiss Vector Search. We evaluate gpu_ext on vector similarity search using the SIFT dataset [26] with an IVF4096,Flat index, scaling from 20M (9.5GB) to 50M (24GB) and 100M (48GB, exceeding GPU memory). IVF indexes have a two-level structure: frequently-accessed centroids and larger posting lists, and Faiss operations have two distinct phases: BUILD performs sequential scans while SEARCH issues random accesses. The Faiss policy in gpu_ext reduces build time by 21–29% on 50M, scaling to 27–40% on 100M as memory pressure grows. For search workloads on SIFT100M, this policy reduces latency by 10–16% across different `nprobe` settings. As shown in Figure 11, the converged policy (~890 LOC: sequential prefetch 375 LOC + stride prefetch 472 LOC + device-side 45 LOC) prefetches posting lists sequentially for oversubscribed datasets and uses stride prediction for K-means iterations during index construction; device-side eBPF programs also trigger prefetching of corresponding posting lists.

The agent tested 8 policies over 5 iterations. Initially, a momentum-based phase detector with cycle eviction improved BUILD but degraded SEARCH, as forward drift in search queries confused the detector. Fixing the classifier revealed eviction policy rather than phase detection controlled performance. Version three, pairing the corrected detector with default eviction, converged. A subsequent fast-path optimization introduced a state-machine bug (policy stuck in SEARCH mode), detected and fixed in one iteration. Only 2 of 8 policies yielded positive results.

5.2.3 Multi-Tenant Memory, Bandwidth, and Scheduling.

We evaluate gpu_ext on multi-tenant GPU workloads, comparing per-tenant policies against global and framework-managed approaches. The tenants share an unpartitioned GPU managed by a system-wide policy.

Compute-bound Scheduler (LC + BE). We evaluate gpu_ext's eBPF struct_ops scheduling policies on *compute-bound* (SM-limited) multi-tenant workloads, without memory oversubscription. 2 latency-critical (LC) processes and 4 best-effort (BE) processes run concurrently, each with 4 CUDA streams submitting 50 compute kernels. We compare the native scheduler against a gpu_ext preemption policy equivalent to GPREEMPT's [13] priority-based timeslice scheduling, with differentiated driver-controlled timeslices (LC: 1s, BE: 200μs), implemented as an eBPF program (925 LOC) without unsafe modification to the driver source. This scheduling policy reduces LC P99 launch latency

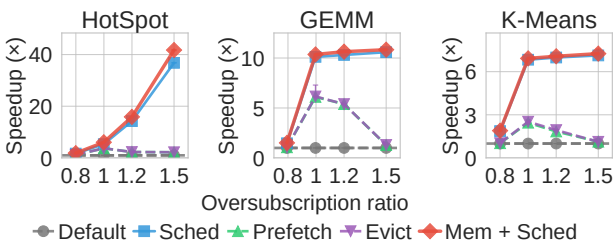


Figure 12. High-priority completion speedup over Default as the oversubscription ratio increases. Points show medians of five paired speedups, and whiskers show their ranges.

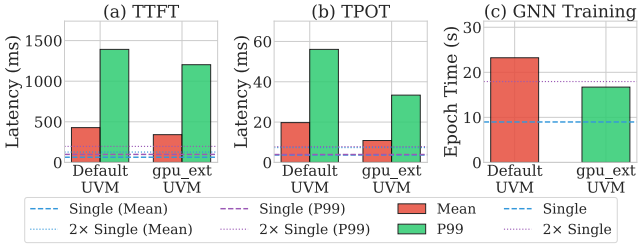


Figure 13. Two-tenant co-location: `llama.cpp` inference (LC) + GNN training (BE).

by 96% (1188μs → 53μs). BE throughput remains nearly unchanged at 11.38 kernels/s with the native scheduler and 11.47 kernels/s with this policy (+0.8%).

Memory-bound Priority Differentiation. We evaluate gpu_ext's memory and scheduling policies for priority differentiation using HotSpot [6] (spatial locality), GEMM from PolyBench [18] (compute-intensive, memory-bound under UVM oversubscription), and K-Means [19] (sparse access). Each experiment runs two concurrent processes competing for GPU memory, with the oversubscription ratio defined as their combined allocation divided by GPU memory capacity. We compare default execution (Default), scheduling alone (Sched), priority-based prefetching (Prefetch, 454 LOC), prefetching with priority-based eviction (Evict, 472 LOC), and their combination with scheduling (Mem + Sched).

Both memory policies and scheduling alleviate the slowdown from memory contention, with scheduling yielding larger improvements in these workloads (Figure 12). Combining memory priority with scheduling further reduces high-priority completion time by 9.5–12.0% for HotSpot, 2.3–3.1% for GEMM, and 1.2–1.5% for K-Means relative to scheduling alone across the evaluated ratios.

Two-Tenant: High-Priority LLM Inference + Background Training. We run two tenants sharing a single GPU: T1 is a high-priority `llama.cpp` inference service (LC workload) serving 100 ShareGPT prompts at 0.2 RPS, and T2 is a background PyTorch GNN training job (BE workload) with 8M nodes requiring 36GB peak memory (oversubscribed). We compare default UVM (driver FIFO/LRU) against gpu_ext per-tenant memory and scheduler policies (~926 LOC, combining Quota LRU, Tree-based Prefetch, and Dynamic Timeslice). Figure 13 shows mutual improvement: LC TPOT drops by 40–45% and TTFT by 14–20%, while GNN training simultaneously improves by 28%, approaching the ideal fair share baseline. This shows the combined memory and scheduling policies in gpu_ext reduce contention for both workloads rather than favoring one over the other.

Summary. With gpu_ext, AI agents can develop policies that translate application access patterns and task priorities into coordinated decisions about execution, data residency, and transfer timing, enabling application needs to guide resource management across memory and scheduling.

5.2.4 Policies from Prior Systems.

We evaluate the seven policies in Table 2.

Methodology. In Figure 15, we compare the workload baseline with the native policy to measure policy benefit. We then compare the gpu_ext and native implementations of the same policy to measure the cost of gpu_ext, keeping the workload and underlying execution unchanged. Unless stated otherwise, we report medians with confidence intervals.

Results. Against the baselines in Figure 15(a–g), MoE-Infinity lowers TTFT by 22.1% (native) and 22.5% (gpu_ext). Expert Buffering cuts full expert loads by 12.85% in both implementations, with 0.7% gpu_ext overhead. The FineMoE policy in gpu_ext raises throughput by 24.6%, with a 0.21% difference from native (95% CI [−0.17%, +0.52%]). Hummingbird lowers LC response p99 by 7.5–8.2% (native) and 5.9–9.1% (gpu_ext). POD-Attention reduces operator latency from 5.360 ms to 4.322 ms (native) and 4.344 ms (gpu_ext). Across shapes, operator latency with the POD-Attention policy in gpu_ext ranges from 0.44% lower to 1.18% higher than native. XSched lowers LC p99 latency from submission to first block execution from 76.9 s to 27.0 s (native) and 27.3 s (gpu_ext). GPREEMPT lowers LC response p99 from 1.796 ms to 1.615 ms (native) and 1.610 ms (gpu_ext).

5.3 RQ2: Mechanism Cost

Verification Overhead. Load-time verification takes 141 ms for `kernelretsnoop` and 12 ms for `threadhist`.

Host Runtime Overhead. We quantify the overhead of the gpu_ext driver runtime independent of policy logic using GEMM from PolyBench [18] and HotSpot [6] (1.1× oversubscription, 35 GB working set) on RTX 5090. With hooks enabled but no policy attached, the overhead is less than 0.2%.

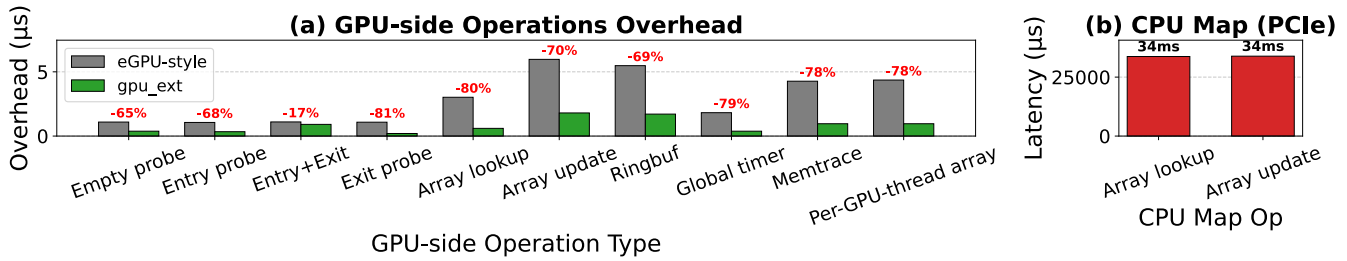


Figure 14. Device-side overhead. (a) Naive per-thread injection vs gpu_ext's SIMT-aware execution. (b) CPU map access latency via PCIe vs GPU-side operations.

Table 2. Prior policies, their underlying mechanisms, and the gpu_ext implementations evaluated in Figure 15(a–g).

System	Original policy	Original and gpu_ext implementation
MoE-Infinity [60]	Uses predicted expert accesses to prioritize prefetching and choose which experts to evict.	The original system uses a user-space runtime to manage expert transfers. gpu_ext implements cache decisions as host-side programs.
Expert Buffering [24]	Chooses which experts to cache, evicting those unused by the current batch first.	The original system uses a user-space cache to manage GPU buffers and CPU-to-GPU copies. gpu_ext implements admission and eviction as host-side programs.
FineMoE [63]	Dynamically chooses which predicted experts to load before they are needed.	The original system uses a user-space cache with CUDA runtime transfers. gpu_ext implements dynamic expert selection as a host-side program.
Hummingbird [23]	Runs smaller background kernels during foreground idle intervals and manages memory by task priority.	The original system schedules split kernels on the host, remaps thread-block coordinates on the device, and modifies UVM for memory management. The gpu_ext policy pairs host-side task admission with device-side block-coordinate mapping.
POD-Attention [29]	Assigns prefill or decode work to each SM according to the desired balance between the two.	The original system selects work inside the fused GPU kernel. In gpu_ext, the host prepares launches and a device-side program selects work within the unchanged fused kernel.
XSched [49]	Runs the highest-priority queued task, preempting lower-priority work when needed.	The original system controls queues on the host, adds device-side kernel-entry checks at Level 2, and uses driver interfaces at Level 3. The gpu_ext policy assigns queue selection to host programs and kernel-entry decisions to device programs.
GPREEMPT [13]	Uses task priority to decide which tasks may run and how much GPU time each receives.	The original system controls task blocking and release in user space and modifies the GPU driver to control timeslices. gpu_ext implements these decisions with programs in user space and the GPU driver.

Device-side Overhead. We compare device-side observability tools built on gpu_ext with NVBit [55] (Figure 16) using llama.cpp prefill on two GPUs: the P40 (Llama 1B, sequence-mixed) and the RTX 5090 (TinyLlama-1.1B, 512-token prefill). On the device, `kernelretsnoop` (153 LOC) records GPU kernel return timestamps, while `threadhist` (89 LOC) counts thread exits to detect load imbalance. `launchlate` (347 LOC) combines host- and device-side observations to measure GPU kernel launch latency. Overhead is measured as token/s degradation during prefill. On the P40, gpu_ext achieves 3–14% overhead versus NVBit's 85–93%, due to the warp-uniform execution model. In panel (b), the three tools incur 5.57%, 2.97%, and 0.22% overhead with gpu_ext, compared with 99.62%, 10.35%, and 8.80% with NVBit, respectively. Attaching to a running process takes 273 ms on average from `bpftime trace` startup to first GPU kernel detection, without application restart.

Device-side Runtime Optimization. We evaluate device-side gpu_ext mechanisms using a vector-add CUDA GPU kernel (32 elements, 10K iterations) from cuda-sample [40] on RTX 5090. Figure 14(a) compares gpu_ext's SIMT-aware warp-level execution against eGPU [62] that injects eBPF bytecode without SIMT-specific verification and optimizations. The baseline incurs overhead due to per-thread execution and uncoalesced memory access,

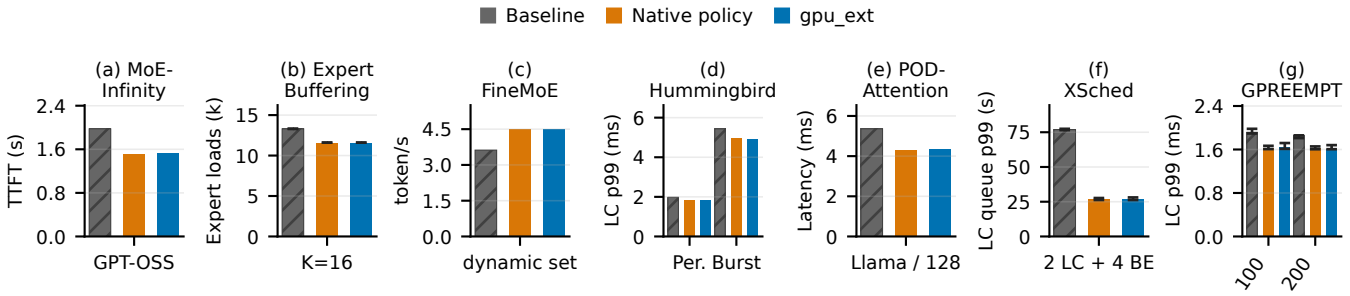


Figure 15. Policy comparisons. Baselines are (a) prediction-disabled expert caching, (b) fixed-K FIFO, (c) prefetching all positively predicted experts, (d,g) default scheduling, (e) two-stream FlashAttention, and (f) native CUDA. In (d), Per. and Burst denote periodic and BurstGPT [56] arrivals. Panel (e) uses Llama decode batch 128. Bars show medians, whiskers show full ranges. Panel (g) uses VGG19 LC at 100 requests/s with ResNet152 BE at 100 or 200 requests/s.

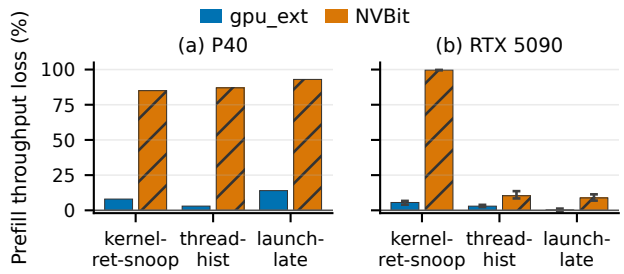


Figure 16. Prefill throughput loss relative to an uninstrumented baseline. (a) P40, Llama 1B. (b) RTX 5090, TinyLlama-1.1B.

while gpu_ext's warp-uniform execution and coalesced map access reduce overhead by approximately 60–80% for most operations. For an arithmetic test kernel, trampoline overhead stays within 1.2–2.2 μs as active warps increase 8×, while per-thread map-update cost grows roughly proportionally. Figure 14(b) shows CPU map access via PCIe is 6000× slower than GPU-side operations, motivating gpu_ext's hierarchical map design that keeps hot state on GPU. We do not compare policy-level performance against the prior workshop version or Neutrino because they are limited to read-only observability (§2).

6 Related Work

`sched_ext` [54], `cache_ext` [68], and XDP extend Linux subsystems via synchronous eBPF callbacks at fixed decision points; gpu_ext generalizes this to asynchronous, cross-device effects on user-defined events (§3.2). Driver-level GPU resource managers [13, 21, 30, 31, 58, 65] require unsafe OS kernel driver modifications and service interruptions to change policy. User-space frameworks [7, 10, 17, 38, 49, 51] offer programmability but lack cross-tenant visibility and privileged hardware control (§2.3). gpu_ext offers runtime-programmable policies without either limitation. NVBit [55] and Neutrino [25] inject instrumentation into GPU binaries for device-side visibility, but lack safety guarantees and target offline profiling rather than online policy enforcement. A prior workshop paper [62] extends eBPF to GPU instrumentation but remains limited to read-only observability with high overhead (§5) due to per-thread scalar execution on SIMT hardware; gpu_ext introduces SIMT-aware verification and warp-level execution for safe, low-overhead policy enforcement.

7 Conclusion

We presented gpu_ext, a policy runtime that models each GPU resource as an asynchronous state machine whose state transitions are directed by verified eBPF programs across the host-device boundary. Across four workloads, gpu_ext policies improve throughput by up to 1.76× over application-tuned baselines and reduce tail latency by up to 2×, demonstrating that OS kernel extensibility extends effectively to GPU management. For seven published resource-management policies, the metrics in Figure 15 differ by less than 2% between gpu_ext and native implementations.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful feedback. Williams' group is supported in part by the National Science Foundation (NSF) under Award #2236966 and by a grant from the eBPF Foundation. This work is also supported in part by NSF Award #2530909 and by NEUTC.

We used Claude and Codex as generative AI tools for coding assistance and language editing.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA,

117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[2] AlphaEvolve team. 2025. AlphaEvolve: A Gemini-powered Coding Agent for Designing Advanced Algorithms. Google DeepMind Blog. https://deepmind.google/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/ Accessed: October 11, 2026.

[3] anon8231489123. 2023. ShareGPT Vicuna Unfiltered Dataset. Hugging Face dataset. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered File: `ShareGPT_V3_unfiltered_cleaned_split.json`. Accessed: October 11, 2026.

[4] Pratheek B, Guilherme Cox, Jan Vesely, and Arkaprava Basu. 2024. SUV: Static Analysis Guided Unified Virtual Memory. In *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO '24)*. IEEE, Austin, TX, USA, 293–308. doi:10.1109/MICRO61859.2024.00030

[5] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU database systems characterization and optimization. *Proceedings of the VLDB Endowment* 17, 3 (2023), 441–454. doi:10.14778/3632093.3632107

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Austin, TX, USA, 44–54. doi:10.1109/IISWC.2009.5306797

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

[8] Audrey Cheng, Shu Liu, Melissa Pan, Ion Stoica, and ADRS team. 2025. Barbarians at The Gate: How AI is Upending Systems Research. ACM SIGOPS Blog. https://www.sigops.org/2025/barbarians-at-the-gate-how-ai-is-upending-systems-research/ Accessed: October 11, 2026.

[9] Jon Chesterfield. 2025. RFC: SPIRV IR as a vendor agnostic GPU representation. LLVM Discourse. https://discourse.llvm.org/t/rfc-spirv-ir-as-a-vendor-agnostic-gpu-representation/85115 Accessed: October 11, 2026.

[10] Patrick H Coppock, Brian Zhang, Eliot H Solomon, Vasilis Kypriotis, Leon Yang, Bikash Sharma, Dan Schatzberg, Todd C Mowry, and Dimitrios Skarlatos. 2025. LithOS: An operating system for efficient machine learning on GPUs. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3731569.3764818

[11] Yuran Ding, Xinwei Chen, Xiaofan Zhang, and Zongwei Zhou. 2025. ASAP: an Agentic Solution to Auto-optimize Performance of Large-Scale LLM Training. In *ML for Systems Workshop at NeurIPS*. San Diego, CA, USA. https://neurips.cc/virtual/2025/loc/san-diego/129091

[12] Rohit Dwivedula, Divyanshu Saxena, Aditya Akella, Swarat Chaudhuri, and Daehyeok Kim. 2025. Man-Made Heuristics Are Dead. Long Live Code Generators! In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks (HotNets)*. Association for Computing Machinery, New York, NY, USA, 51–60. doi:10.1145/3772356.3772413

[13] Ruwen Fan, Tingxu Ren, Minhui Xie, Shiwei Gao, Jiwu Shu, and Youyou Lu. 2025. GPREEMPT: GPU Preemptive Scheduling Made General and Efficient. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, Boston, MA, 263–272. https://www.usenix.org/conference/atc25/presentation/fan

[14] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 224–235. doi:10.1145/3307650.3322224

[15] Georgi Gerganov. 2023. llama.cpp. GitHub repository. https://github.com/ggml-org/llama.cpp Accessed: October 11, 2026.

[16] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. Association for Computing Machinery, New York, NY, USA, 1069–1084. doi:10.1145/3314221.3314590

[17] In Gim, Zhiyao Ma, Seung-seob Lee, and Lin Zhong. 2025. Pie: A Programmable Serving System for Emerging LLM Applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 415–430. doi:10.1145/3731569.3764814

[18] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. IEEE, San Jose, CA, USA, 1–10. doi:10.1109/InPar.2012.6339595

[19] Yongbin Gu, Wenxuan Wu, Yunfan Li, and Lizhong Chen. 2020. UVMBench: A Comprehensive Benchmark Suite for Researching Unified Virtual Memory in GPUs. arXiv:2007.09822 https://arxiv.org/abs/2007.09822

[20] Pouya Hamadanian, Pantea Karimi, Arash Nash-Esfahany, Kimia Noorbakhsh, Joseph Chandler, Ali Parandeh, Mohammad Alizadeh, and Hari Balakrishnan. 2025. Glia: A Human-Inspired AI for Systems Design and Optimization. ACM SIGOPS Blog. https://www.sigops.org/2025/glia-a-human-inspired-ai-for-systems-design-and-optimization/ Accessed: October 11, 2026.

[21] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[22] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 54–66. doi:10.1145/3281411.3281443

[23] Tiancheng Hu, Chenxi Wang, Ting Cao, Jin Qin, Lei Chen, Xinyu Xiao, Junhao Hu, Hongliang Tian, Shoumeng Yan, Huimin Cui, Quan Chen, and Tao Xie. 2026. Hummingbird: SLO-Oriented GPU Preemption at Microsecond-scale. arXiv:2601.04071v2. https://arxiv.org/abs/2601.04071v2

[24] Haiyang Huang, Newsha Ardalani, Anna Sun, Liu Ke, Hsien-Hsin S. Lee, Shruti Bhosale, Carole-Jean Wu, and Benjamin Lee. 2024. Toward Efficient Inference for Mixture of Experts. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, Vol. 37. Curran Associates, Inc., Vancouver, BC, Canada, 84033–84059. doi:10.52202/079017-2670

[25] Songlin Huang and Chenshu Wu. 2025. Neutrino: Fine-grained GPU Kernel Profiling via Programmable Probing. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 331–355. https://www.usenix.org/conference/osdi25/presentation/huang-songlin

[26] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128. doi:10.1109/TPAMI.2010.57

[27] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547. doi:10.1109/TBDATA.2019.2921572

[28] Nathan Jones, Tyler Allen, and Rong Ge. 2025. HELM: Characterizing Unified Memory Accesses to Improve GPU Performance under Memory Oversubscription. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*. Association for Computing Machinery, New York, NY, USA, 490–504. doi:10.1145/3712285.3759812

[29] Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. 2025. POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Association for Computing Machinery, New York, NY, USA, 897–912. doi:10.1145/3676641.3715996

[30] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. USENIX Association, Portland, OR. https://www.usenix.org/conference/usenixatc11/timegraph-gpu-scheduling-real-time-multi-tasking-environments

[31] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. 2012. Gdev: First-Class GPU Resource Management in the Operating System. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 401–412. https://www.usenix.org/conference/atc12/technical-sessions/presentation/kato

[32] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[33] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 49–63. doi:10.1145/3297858.3304044

[34] Mao Lin, Yuan Feng, Guilherme Cox, and Hyeran Jeon. 2025. Forest: Access-aware GPU UVM Management. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 137–152. doi:10.1145/3695053.3731047

[35] Yinnian Lin, Lei Zou, and Xunbin Su. 2024. Towards Sufficient GPU-accelerated Dynamic Graph Management: Survey and Experiment. *Proceedings of the VLDB Endowment* 18, 3 (2024), 599–612. doi:10.14778/3712221.3712228

[36] Yuhan Liu, Jiayi Yao, Yihua Cheng, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. 2025. LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference. arXiv:2510.09665v2 https://arxiv.org/abs/2510.09665v2

[37] Nurlan Nazaraliyev, Elaheh Sadredini, and Nael Abu-Ghazaleh. 2025. DREAM: Device-Driven Efficient Access to Virtual Memory. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. Association for Computing Machinery, New York, NY, USA, 1190–1205. doi:10.1145/3721145.3725748

[38] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency model serving with software-defined GPU scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 595–610. doi:10.1145/3600006.3613163

[39] NVIDIA. 2022. NVIDIA Open GPU Kernel Modules. https://github.com/NVIDIA/open-gpu-kernel-modules Accessed: October 11, 2026.

[40] NVIDIA. 2025. NVIDIA CUDA Samples. https://github.com/NVIDIA/cuda-samples Accessed: October 11, 2026.

[41] NVIDIA. 2026. Work Stealing with Cluster Launch Control — CUDA Programming Guide. https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cluster-launch-control.html Accessed: October 11, 2026.

[42] NVIDIA Corporation. 2026. NVIDIA Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/ Accessed: October 11, 2026.

[43] OpenAI. 2025. gpt-oss-120b & gpt-oss-20b Model Card. arXiv:2508.10925 https://arxiv.org/abs/2508.10925

[44] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *The VLDB Journal* 33, 5 (2024), 1591–1615. doi:10.1007/s00778-024-00864-x

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*, Vol. 32. Curran Associates, Inc., Vancouver, BC, Canada, 8026–8037. https://proceedings.neurips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[46] PREVAIL contributors. 2026. PREVAIL: An eBPF Verifier Based on Abstract Interpretation. https://github.com/vbpf/prevail Accessed: October 11, 2026.

[47] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Association for Computing Machinery, New York, NY, USA, 3505–3506. doi:10.1145/3394486.3406703

[48] John Ravi, Tri Nguyen, Huiyang Zhou, and Michela Becchi. 2021. PILOT: a Runtime System to Manage Multi-tenant GPU Unified Memory Footprint. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, Bengaluru, India, 442–447. doi:10.1109/HiPC53243.2021.00063

[49] Weihang Shen, Mingcong Han, Jialong Liu, Rong Chen, and Haibo Chen. 2025. XSched: Preemptive scheduling for diverse XPUs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 671–692. https://www.usenix.org/conference/osdi25/presentation/shen-weihang

[50] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 https://arxiv.org/abs/1909.08053

[51] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1075–1092. doi:10.1145/3627703.3629578

[52] The Linux Kernel Contributors. 2026. *DRM GPU scheduler*. The Linux Kernel Documentation. https://docs.kernel.org/gpu/drm-mm.html#gpu-scheduler Accessed: October 11, 2026.

[53] The Linux Kernel Contributors. 2026. *Heterogeneous Memory Management (HMM)*. The Linux Kernel Documentation. https://docs.kernel.org/mm/hmm.html Accessed: October 11, 2026.

[54] The Linux Kernel Contributors. 2026. sched_ext: Extensible Scheduler Class. The Linux Kernel Documentation. https://docs.kernel.org/scheduler/sched-ext.html Accessed: October 11, 2026.

[55] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, New York, NY, USA, 372–383. doi:10.1145/3352460.3358307

[56] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2025. BurstGPT: A Real-World Workload Dataset to Optimize LLM Serving Systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '25)*. Association for Computing Machinery, New York, NY, USA, 5831–5841. doi:10.1145/3711896.3737413

[57] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/2851141.2851145

[58] Yidi Wang, Cong Liu, Daniel Wong, and Hyoseung Kim. 2024. GCAPS: GPU Context-Aware Preemptive Priority-Based Scheduling for Real-Time Tasks. In *36th Euromicro Conference on Real-Time Systems (ECRTS 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 298)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 14:1–14:25. doi:10.4230/LIPIcs.ECRTS.2024.14

[59] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

[60] Leyang Xue, Yao Fu, Zhan Lu, Chuanhao Sun, Luo Mai, and Mahesh Marina. 2025. MoE-Infinity: Efficient MoE Inference on Personal Machines with Sparsity-Aware Expert Cache. arXiv:2401.14361v3. https://arxiv.org/abs/2401.14361v3

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

[62] Yiwei Yang, Tong Yu, Yusheng Zheng, and Andrew Quinn. 2025. eGPU: Extending eBPF Programmability and Observability to GPUs. In *Proceedings of the 4th Workshop on Heterogeneous Composable and Disaggregated Systems*. Association for Computing Machinery, New York, NY, USA, 73–79. doi:10.1145/3723851.3726984

[63] Hanfei Yu, Xingqi Cui, Hong Zhang, Hao Wang, and Hao Wang. 2026. Taming Latency-Memory Trade-Off in MoE-Based LLM Serving via Fine-Grained Expert Offloading. In *Proceedings of the 21st European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, USA, 176–191. doi:10.1145/3767295.3769319

[64] Peifeng Yu and Mosharaf Chowdhury. 2020. Fine-grained GPU sharing primitives for deep learning applications. *Proceedings of Machine Learning and Systems* 2 (2020), 98–111. https://proceedings.mlsys.org/paper_files/paper/2020/hash/d9cd83bc91b8c36a0c7c0fcca59228f2-Abstract.html

[65] Shulai Zhang, Ao Xu, Quan Chen, Han Zhao, Weihao Cui, Zhen Wang, Yan Li, Limin Xiao, and Minyi Guo. 2025. Efficient Performance-Aware GPU Sharing with Compatibility and Isolation through Kernel Space Interception. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, Boston, MA, 1003–1019. https://www.usenix.org/conference/atc25/presentation/zhang-shulai

[66] Yusheng Zheng, Yanpeng Hu, Wei Zhang, and Andi Quinn. 2025. Towards Agentic OS: An LLM Agent Framework for Linux Schedulers. In *ML for Systems Workshop at NeurIPS*. San Diego, CA, USA. https://neurips.cc/virtual/2025/loc/san-diego/129092

[67] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, Xiaozheng Lai, Dan Williams, and Andi Quinn. 2025. Extending Applications Safely and Efficiently. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 557–574. https://www.usenix.org/conference/osdi25/presentation/zheng-yusheng

[68] Tal Zussman, Ioannis Zarkadas, Jeremy Carin, Andrew Cheng, Hubertus Franke, Jonas Pfefferle, and Asaf Cidon. 2025. cache_ext: Customizing the Page Cache with eBPF. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 462–478. doi:10.1145/3731569.3764820

A Artifact

The gpu_ext artifact is available at https://github.com/eunomia-bpf/gpu_ext. It includes the system implementation, resource-management policies, benchmark harnesses, and recorded experimental results. The artifact guide provides the paper-to-artifact mapping and instructions for building, running, and reanalyzing the experiments.

The implementation guide links the following validation procedures to their source and retained tests.

A.1 Transition Validation

The pseudocode uses resource-specific routines for synchronization, state validation, fallback, and transition initiation.

```
record(slot, action):
    if not slot.attempted:
        slot = (attempted=true, conflict=false, action)
        return RECORDED
    if slot.conflict: return CONFLICT
    if slot.action == action: return REPEAT
    slot.conflict = true
    return CONFLICT

apply(resource, observed, request):
    with driver_guard(resource):
        current = driver_state(resource)
        if not valid_owner(resource, observed):
            return fallback(resource, request,
                            INVALID_OWNER)
        if not valid_state(observed, current):
            return fallback(resource, request, STALE_STATE)
        if request.conflict:
            return fallback(resource, request, CONFLICT)
        if not request.attempted:
            return fallback(resource, request, NO_REQUEST)
        if not allowed(current, request.action):
            return fallback(resource, request,
                            INVALID_ACTION)
        return initiate_native(resource, request.action)
```

A.2 SIMT Verifier

The analysis state contains register and stack-byte uniformity tags, pointer provenance, and map identities; `transfer` applies instruction and helper semantics, and `join` merges incoming states.

```
verify_gpu(program, maps):
    reject malformed input or unsupported GPU maps
    if PREVAIL(program, GPU helper/map model,
               termination=true) fails: reject
    state = {entry: initial_state()}
    worklist = [entry]
    while worklist is not empty:
        pc = pop(worklist)
        out = transfer(program[pc], state[pc])
        for s in successors(pc):
            new = out if s not in state
                  else join(state[s], out)
            if s not in state or new != state[s]:
                state[s] = new
                enqueue(worklist, s)
    for each reachable instruction and input state:
        require uniform branch operands
        require uniform atomic target addresses
        require uniform map keys and update flags
        require uniform shared-map payloads
        reject prohibited helpers and nonuniform
               checked host-bridge payloads
    accept only if all checks pass
```

Runtime rejection requires a verifier-enabled build in strict mode; warning-only and disabled modes do not enforce rejection. The following pairs were tested through the public GPU-verifier entry point.

Rejected program	Accepted control
Branch on lane ID	Branch on warp ID
Lane-derived map key	Warp-derived map key
Lane-derived shared-map value	Warp-derived shared-map value

Table 3. Paired SIMT rejection and acceptance examples.