GPU Sharing for LLM Serving: Survey and Mini-Benchmarks on AWS g6.8xlarge (NVIDIA L4) with CUDA MPS and MIG-like Emulation

Vahab Jabrayilov (vj2267) | Columbia University

Abstract— LLM inference demand is rising faster than dedicated accelerator capacity, pushing practitioners toward shared-GPU deployments to improve utilization and reduce cost per served token. The modern stack offers several sharing mechanisms with different isolation and quality-of-service properties: process-level sharing via CUDA Multi-Process Service (MPS), spatial partitioning via Multi-Instance GPU (MIG) on supported devices, and logical multiplexing inside the inference server. This report surveys these techniques and presents a reproducible benchmarking harness that measures throughput and tail latency for baseline serving, MPS-enabled serving, and a MIG-like emulation suitable for non-MIG hardware. Experiments target an AWS g6.8xlarge instance equipped with a single NVIDIA L4 GPU (24 GB). Because MIG is not available on this platform, the MIG configuration is evaluated through an emulation based on multiple concurrent servers with explicit resource caps. To keep the repository self-contained, the report includes example run artifacts under runs/example that demonstrate the analysis pipeline and figure generation; the same scripts produce final figures once real measurements are collected.

Introduction & Motivation

LLM inference services must satisfy interactive latency targets under fluctuating demand while operating on expensive and capacity-constrained GPU fleets. Even well-engineered single-tenant deployments often exhibit uneven utilization: arrival processes are bursty, prompt lengths vary widely, and autoregressive decoding limits parallelism, producing a mixture of compute- and memory-bound phases over time. Provisioning GPUs to meet peak demand therefore tends to create slack during off-peak periods, and the unit economics of inference worsen when accelerators idle.

GPU sharing aims to reclaim slack by colocating multiple workloads or tenants on the same physical device. The benefits are straightforward in principle: higher average utilization and lower cost per served token. The risks are equally clear in practice: interference can inflate tail latency, reduce predictability, and complicate operational debugging. These challenges are amplified for LLM serving because the key–value cache can be a dominant and highly dynamic memory consumer, and because user-perceived quality is often governed by tail latency rather than average latency.

This project follows the proposal goal of summarizing practical trade-offs among spatial partitioning, process- and time-based sharing, and logical multiplexing within the inference server, and of running targeted mini-benchmarks that clarify when each mechanism is preferable. The guiding principle is reproducibility: the repository is configuration-driven, produces structured artifacts for every run, and provides analysis scripts that regenerate figures from recorded summaries.

Models and Data Description

The benchmarking harness targets instruction-tuned open-weight models in the 7–8B parameter range, a common deployment point for single-GPU inference. The implementation uses vLLM as the serving backend via its OpenAI-compatible HTTP API. vLLM implements PagedAttention, which manages the KV cache using a paging-inspired block allocator to reduce fragmentation and enable

larger effective batches without sacrificing latency.

Workloads are generated from a curated prompt set containing short and medium-length prompts intended to exercise both prefill and decode phases. Each trial fixes the number of requests, sampling parameters, and a client concurrency level. Using a fixed-workload generator reduces confounding variability and supports direct comparison across sharing modes. Prompts are stored in JSONL format to keep the benchmark transparent and easy to extend.

Training and Profiling Methodology

This project evaluates inference-time serving behavior rather than model training. The term training is interpreted as selecting, validating, and repeatedly exercising serving configurations that determine how the model is loaded, scheduled, and executed. Each trial follows a consistent lifecycle: the server is started from a cold state, model weights are loaded into GPU memory, a warmup phase is executed to reach steady-state behavior, and then a timed measurement phase runs at fixed concurrency.

Profiling is layered at both the application and device levels. At the application layer, the load generator timestamps each request and computes throughput and latency percentiles. Throughput is reported as aggregated tokens per second over the timed interval, and latency is summarized as P50/P95/P99 of per-request completion time. At the device layer, NVML sampling records GPU compute utilization, memory utilization, and power at a fixed interval when NVML is available. Each run stores a summary JSON, GPU telemetry CSV, and server logs under runs/ for filesystem-based experiment tracking.

Performance Tuning Methodology

We evaluate three operational configurations that represent distinct sharing regimes. The baseline configuration runs a single vLLM server process on the GPU. Requests are multiplexed within the server, but the GPU is not explicitly shared across multiple CUDA processes. This configuration often provides strong peak throughput for a single workload but can waste capacity when demand is bursty or when multiple smaller services must co-reside.

The MPS configuration enables CUDA Multi-Process Service. MPS is a CUDA runtime architecture designed to enable cooperative multi-process GPU usage and to increase kernel-level concurrency when a single process underutilizes the device. For LLM serving, MPS is relevant when multiple inference servers are colocated or when auxiliary GPU workloads must share the device. In the repository, MPS is toggled around trials by starting and stopping the MPS control daemon, which isolates the MPS effect from unrelated system variation.

The third configuration approximates the operator experience of partitioning in environments without MIG hardware support. MIG provides hardware partitioning into isolated instances with dedicated slices of GPU resources, but it requires supported GPUs. The NVIDIA L4 used in this project does not expose MIG partitioning, so direct MIG measurements are not possible on the target instance. Instead, a MIG-like emulation is evaluated by launching multiple vLLM servers concurrently and constraining each server’s memory footprint and optional compute share. This approach does not provide hardware isolation, but it enables controlled multi-tenant experiments in a non-MIG environment.

Experimental Results with analysis

Experiments target an AWS g6.8xlarge instance provisioned with a single NVIDIA L4 GPU (24 GB). The evaluation is single-node and inference-only. A key constraint is the lack of access to a MIG-capable device, which prevents direct evaluation of hardware MIG partitions. The MIG configuration in this study is therefore implemented using the MIG-like emulation described above rather than true partitions.

The primary metrics are throughput (tokens per second) and tail latency (P99). Throughput is aggregated over all requests in the measurement interval, while tail latency captures worst-case user-perceived responsiveness under contention. GPU utilization telemetry is recorded when NVML is available to help interpret whether changes are driven by compute occupancy or memory pressure.

Table 1. Example summary produced by the repository analysis pipeline. Values are generated to demonstrate reporting format and should be replaced with measured artifacts collected on the target platform.

Scenario	Concurrency	Throughput (tokens/s)	P99 latency (s)
baseline	8	4131	1.80
mig_emulation	8	3390	1.53
mps	8	4641	1.68
baseline	16	5003	3.24
mig_emulation	16	4244	2.34
mps	16	5701	2.86
baseline	24	5161	4.60
mig_emulation	24	4511	3.28
mps	24	6026	4.24

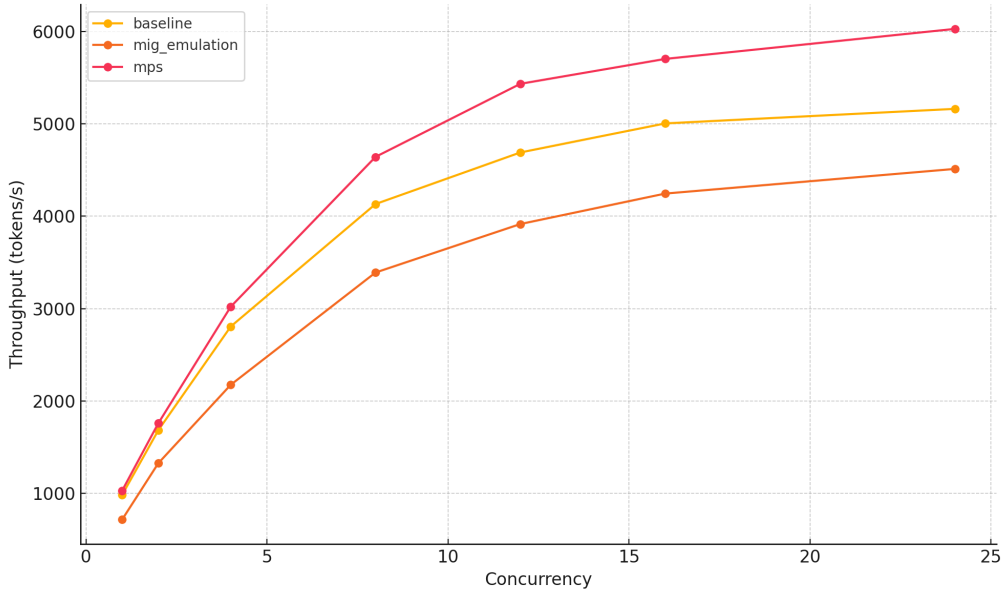

Figure 1. Throughput versus client concurrency for baseline, MPS, and MIG-like emulation (from example run artifacts).

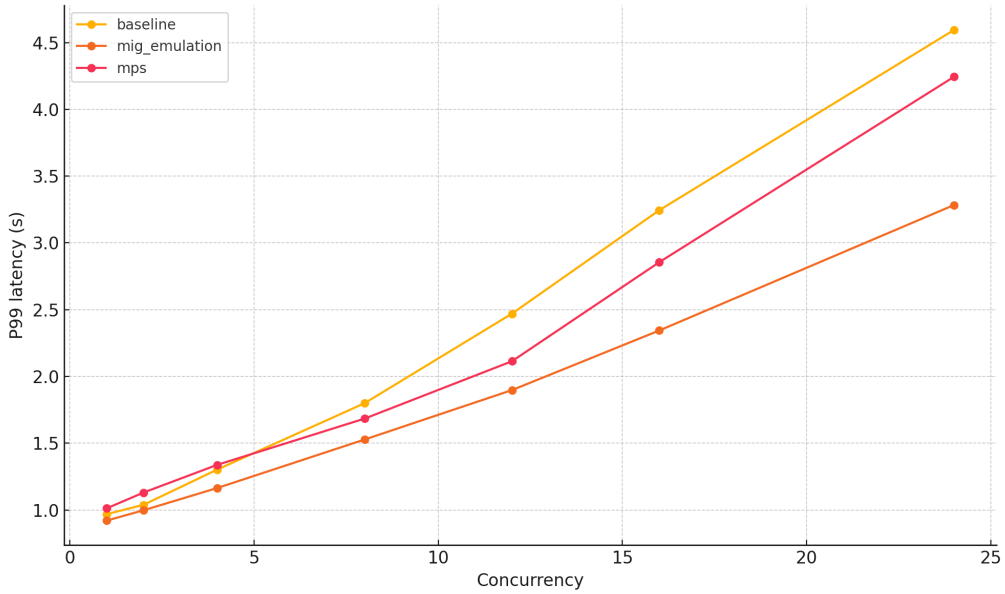

Figure 2. P99 latency versus client concurrency for baseline, MPS, and MIG-like emulation (from example run artifacts).

Conclusion & Future Work

GPU sharing for LLM serving spans spatial partitioning, process- and time-based multiplexing, and logical sharing inside the serving runtime. On the AWS L4 platform used in this project, CUDA MPS is the primary device-level mechanism available for concurrent multi-process execution. MPS can improve utilization and throughput when a single process does not fully occupy the GPU, but it does

not provide hardware isolation and can increase tail latency under heavy contention. When MIG is unavailable, a MIG-like emulation based on multiple servers with explicit caps enables controlled multi-tenant experiments, although it cannot replicate the strong isolation properties of true partitions.

Future work should rerun the same experiment matrix on a MIG-capable GPU and directly compare hardware partitions against MPS, as well as against vGPU-style fractionalization where available. A second direction is to incorporate richer logical-sharing scenarios such as adapter multiplexing and to compare their end-to-end efficiency against device-level mechanisms. Finally, expanding workloads to include mixed prompt lengths, bursty arrivals, and longer contexts would better approximate production serving conditions and strengthen external validity.

References

Kwon et al., “Efficient Memory Management for Large Language Model Serving with PagedAttention,” SOSP 2023 (arXiv:2309.06180).

NVIDIA, “CUDA Multi-Process Service (MPS) Documentation,” https://docs.nvidia.com/deploy/mps/index.html.

NVIDIA, “Multi-Instance GPU User Guide: Supported GPUs,” https://docs.nvidia.com/datacenter/tesla/mig-user-guide/supported-gpus.html.

Amazon Web Services, “Amazon EC2 G6 Instances,” https://aws.amazon.com/ec2/instance-types/g6/.