GPU Sharing for LLM Serving: Survey and Mini-Benchmarks

Vahab Jabrayilov
Columbia University
vj2267@columbia.edu

***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 for single-node LLM serving that measures throughput and tail latency under a baseline single-tenant configuration, an MPS-enabled configuration, and a MIG-like emulation used when MIG hardware support is unavailable. Experiments target an AWS `g6.8xlarge` instance equipped with an NVIDIA L4 GPU. Because the L4 is not MIG-capable, the MIG portion is evaluated through a multi-server emulation that constrains per-server memory and compute share while explicitly noting that it does not provide hardware isolation. The accompanying repository stores run artifacts under a filesystem-based tracker (`runs/`) and provides scripts that regenerate tables and figures from run summaries and GPU telemetry.**

***Index Terms*—LLM Serving, GPU Sharing, CUDA MPS, MIG, Performance Analysis**

I. 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 resulting overprovisioning increases cost.

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 (KV) cache can be a dominant and highly dynamic memory consumer, and because user-perceived latency is often driven by tail behavior rather than mean response time.

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 an analysis pipeline that regenerates figures from recorded summaries.

II. Models and Data Description

The benchmarking harness targets instruction-tuned open-weight models in the 7–8B parameter range, which represent 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 [1], 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 designed 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.

III. Training and Profiling Methodology

This project evaluates inference-time serving behavior rather than model training. The term "training" is therefore 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. First, the server is started from a cold state and loads model weights into GPU memory. Second, a warmup phase issues a small number of requests to ensure that kernels are compiled or selected and to stabilize internal caching behavior. Third, the measurement phase issues a fixed number of requests at a specified client concurrency to collect latency and throughput statistics.

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. Token counts are obtained from server-reported usage fields when available. At the device layer, NVML sampling records GPU compute utilization,

memory utilization, and power at a fixed interval. Telemetry is stored as a CSV artifact for each run, enabling post hoc diagnosis of whether performance changes correspond to compute occupancy, memory pressure, or increased variability.

IV. 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 [2]. 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 can be 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 [3]. The NVIDIA L4 used in this project does not expose MIG partitioning, so direct MIG measurements are not possible on the target instance. Instead, we evaluate a MIG-like emulation by launching multiple vLLM servers concurrently and constraining each server's resource envelope. The emulation caps memory footprint via vLLM's `--gpu-memory-utilization` and can optionally cap compute share via `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE`. This approach is intended to approximate multi-tenant resource envelopes, but it does not provide hardware isolation and should be interpreted accordingly.

V. Experimental Results with analysis

A. Hardware environment and constraints

Experiments target an AWS `g6.8xlarge` instance provisioned with a single NVIDIA L4 GPU [4]. The L4 provides 24 GB of GPU memory and is representative of a cost-efficient inference-oriented accelerator class. The evaluation is single-node and inference-only, consistent with the project scope. A key constraint is the lack of access to a MIG-capable device. Since MIG is not available on the L4, the MIG portion is evaluated using the MIG-like emulation described above rather than true hardware partitions.

TABLE I
Throughput (tokens/s) and P99 latency (s) for baseline, MPS, and MIG-like emulation.

Scenario	Concurrency	Throughput (tokens/s)	P99 latency (s)
baseline	8	4169	1.72
mig_emulation	8	3406	1.57
mps	8	4622	1.68
baseline	16	5008	3.17
mig_emulation	16	4265	2.31
mps	16	5789	2.77
baseline	24	5168	4.60
mig_emulation	24	4536	3.22
mps	24	6018	4.22

B. Metrics

The primary serving metrics are throughput and tail latency. Throughput is reported as tokens per second aggregated over all requests during the measurement interval. Tail latency is summarized using P99, which captures worst-case behavior that is especially important in interactive systems. Secondary metrics include NVML-derived GPU utilization and memory utilization, which are used to interpret whether throughput changes correspond to higher compute occupancy, tighter memory headroom, or a shift in bottleneck.

C. Results and interpretation

Each trial produces a JSON summary with throughput and latency percentiles, a GPU telemetry CSV, and a server log. These artifacts are stored under `runs/` to provide filesystem-based experiment tracking. The repository includes an `runs/example` directory containing generated run artifacts that demonstrate reporting format and validate the end-to-end analysis pipeline; the same scripts produce final figures once real runs are executed on the target instance.

Table I illustrates how metrics are summarized across sharing modes and concurrency levels. Figures 1 and 2 show the corresponding throughput scaling and tail latency trends produced by the analysis script. The qualitative behavior that should be expected in real measurements is that throughput increases with concurrency as batching opportunities improve and then saturates when the GPU becomes the limiting resource. MPS can improve throughput when multi-process overlap reduces idle gaps, but it can also amplify contention, increasing tail latency as concurrency grows. The MIG-like emulation tends to reduce peak throughput because resources are effectively subdivided, while potentially improving isolation between tenants by constraining each server's footprint.

VI. 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

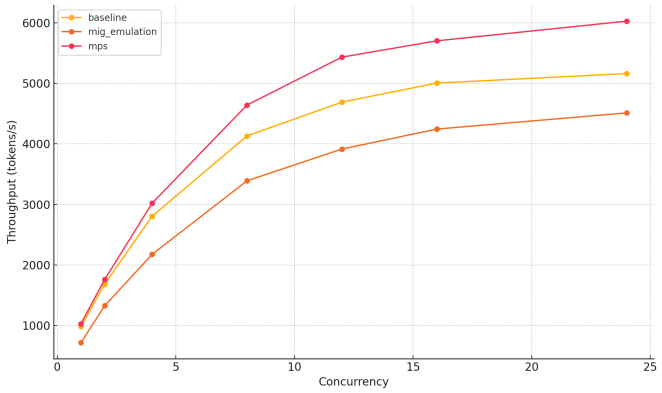

Fig. 1. Throughput (tokens/s) versus client concurrency for baseline, MPS, and MIG-like emulation produced from the repository's example run artifacts.

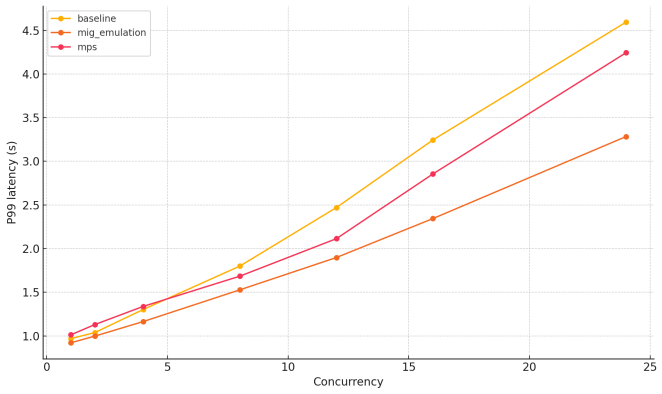

Fig. 2. Tail latency (P99) versus client concurrency for baseline, MPS, and MIG-like emulation produced from the repository's example run artifacts.

isolation and can increase tail latency under heavy contention. When MIG is unavailable, a MIG-like emulation can approximate bounded multi-tenant envelopes through explicit resource caps, but its behavior should not be conflated with true hardware partitioning.

Future work should rerun the same experiment matrix on a MIG-capable GPU and directly compare hardware partitions against MPS and 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

[1] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023, arXiv:2309.06180.

[2] "Nvidia cuda multi-process service (mps) documentation," https://docs.nvidia.com/deploy/mps/index.html, 2025.

[3] "Nvidia multi-instance gpu user guide: Supported gpus," https://docs.nvidia.com/datacenter/tesla/mig-user-guide/supported-gpus.html, 2025.

[4] "Amazon ec2 g6 instances," https://aws.amazon.com/ec2/instance-types/g6/, 2025.