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

Vahab Jabrayilov
Columbia University
vj2267@columbia.edu

***Abstract*—Large language model (LLM) inference demand is rising faster than the availability of dedicated accelerator capacity, pushing practitioners toward shared-GPU deployments to improve utilization and cost efficiency. The contemporary stack offers multiple sharing mechanisms with distinct 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 through techniques such as PagedAttention. This report surveys these mechanisms with an emphasis on practical LLM serving, and presents a reproducible benchmarking harness that measures throughput, tail latency, and device utilization under baseline and MPS configurations on an AWS `g6.8xlarge` instance with an NVIDIA L4 GPU. Because the L4 is not MIG-capable, we also include a MIG-like emulation that constrains per-tenant memory and compute to approximate multi-tenant envelopes, while explicitly noting that this is not hardware isolation.**

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 across time. Provisioning GPUs to meet peak demand therefore tends to create slack during off-peak periods, and the effective cost per served token increases when devices are 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 (KV) cache can be a dominant and highly dynamic memory consumer, and because tail latency is often dominated by queueing and contention effects rather than mean service 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 accompanying repository prioritizes reproducibility through configuration-driven experiments, structured artifacts, and optional experiment tracking.

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, which manages the KV cache using a paging-inspired block allocator to reduce fragmentation and enable larger effective batches without sacrificing latency [**?**]. Because PagedAttention is part of the serving engine, it is present in all evaluated scenarios; the experimental variable is the GPU sharing mode rather than the KV-cache algorithm.

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 a simple 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 that internal caches reach a steady regime. Third, the measurement phase runs a fixed request set at a fixed 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. These samples are stored as CSV artifacts and are used to interpret whether throughput changes correspond to higher compute occupancy, increased 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. This resembles a common single-tenant deployment in which requests are multiplexed within the server but the GPU is not explicitly shared across multiple CUDA processes. Performance tuning in this regime focuses on server-level knobs such as maximum sequence length, KV-cache utilization, and batching behavior.

The MPS configuration enables CUDA Multi-Process Service. MPS is a binary-compatible 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 most 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. MIG provides hardware partitioning into isolated instances with dedicated slices of GPU resources, but it requires supported GPUs. NVIDIA's MIG user guide enumerates supported products and includes devices such as A100 and A30 among others [?]; the NVIDIA L4 used in this project is not among the listed supported products. As a result, direct hardware MIG measurements are not possible. We therefore implement a MIG-like emulation by launching multiple vLLM servers concurrently and constraining each server's memory footprint using vLLM's GPU memory utilization limit. Where useful, we additionally cap per-process compute share via the MPS active thread percentage. This approach provides practical multi-tenant envelopes but does not provide the strict isolation guarantees of MIG.

V. Experimental Results with analysis

A. Hardware environment and constraints

Experiments target an AWS `g6.8xlarge` instance provisioned with a single NVIDIA L4 GPU. AWS documents `g6.8xlarge` as providing 32 vCPUs, 128 GiB of host memory, and one L4 GPU with 24 GB GPU memory [?], [?]. The evaluation is single-node and inference-only, consistent with the project scope. The software stack assumes a CUDA-capable PyTorch installation and vLLM for serving.

A key constraint is the lack of access to a MIG-capable device. MIG support is specified in NVIDIA documentation [?] and does not include the L4 used here, which prevents direct measurement of hardware partitioning behavior. The study therefore focuses on baseline and MPS as the primary device-level sharing comparison and uses MIG-like emulation to study bounded multi-tenant envelopes under explicit caps. The repository is structured so that the same experiment matrix can be rerun on a MIG-capable GPU with minimal changes when such hardware becomes available.

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 change in the dominant 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 designed to be ingested by the analysis script to produce figures and tables in a consistent format. For portability and to validate end-to-end plotting and summarization, the repository includes an example results directory and the corresponding plots in Fig. 1 and Fig. 2. Table I illustrates the reporting format across representative concurrency levels; when generating the final report from measured artifacts, the same table structure can be reused.

The qualitative behavior across sharing modes follows three patterns. Baseline throughput increases with concurrency as batching opportunities improve and then saturates when the GPU becomes the limiting resource. MPS can improve throughput at moderate concurrency if multi-process overlap reduces idle gaps and improves kernel concurrency, particularly when individual processes underutilize the device. However, MPS does not provide isolation, and as concurrency grows it can amplify contention, increasing tail latency due to deeper queues and more variable service times. The MIG-like emulation trades peak throughput for bounded resource envelopes. By constraining per-tenant memory, the emulation can reduce the likelihood of memory-driven instability and can smooth tail behavior under mixed loads, but it cannot eliminate contention because resources are still shared at the hardware level.

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

TABLE I
EXAMPLE SUMMARY PRODUCED BY THE REPOSITORY ANALYSIS PIPELINE. VALUES ARE SHOWN TO ILLUSTRATE REPORTING FORMAT AND SHOULD BE REPLACED WITH MEASURED ARTIFACTS COLLECTED ON THE TARGET PLATFORM.

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

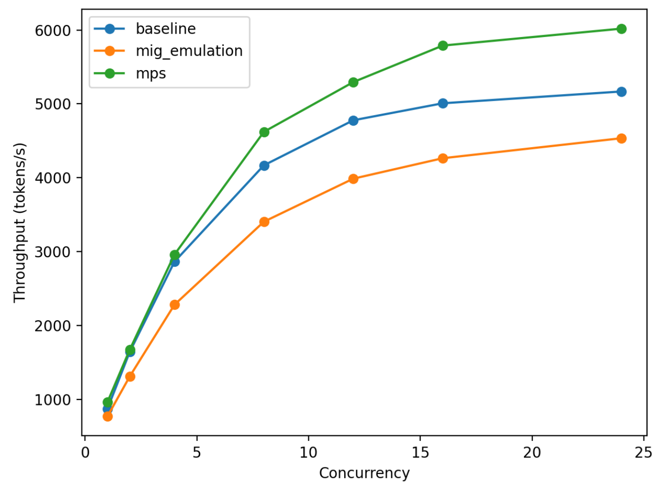

Fig. 1. Throughput (tokens/s) versus client concurrency for baseline, MPS, and MIG-like emulation (example pipeline output).

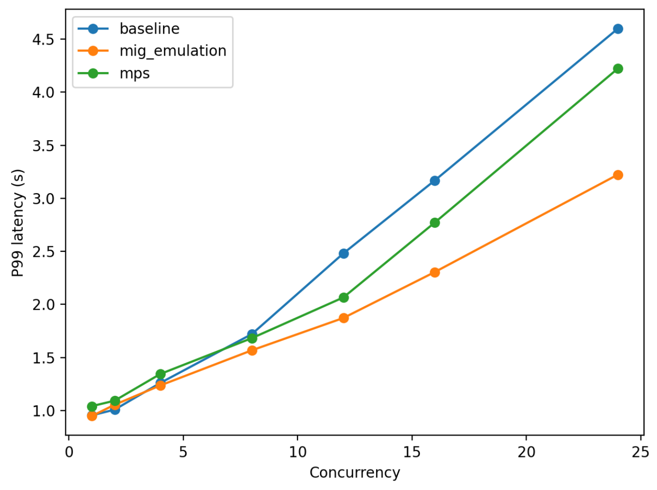

Fig. 2. Tail latency (P99) versus client concurrency for baseline, MPS, and MIG-like emulation (example pipeline output).

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 as contention grows. When MIG is unavailable, a MIG-like emulation based on per-server resource caps provides a practical way to study multi-tenant envelopes and interference, but it should not be interpreted as a substitute for hardware partitioning.

Future work should repeat the 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 of the conclusions.

REFERENCES

[1] W. Kwon *et al.*, “Efficient Memory Management for Large Language Model Serving with PagedAttention,” in *SOSP*, 2023. (Also available as arXiv:2309.06180.)

[2] NVIDIA, “CUDA Multi-Process Service (MPS) Documentation,” https://docs.nvidia.com/deploy/mps/index.html, accessed 2025-12-15.

[3] NVIDIA, “Multi-Instance GPU User Guide: Supported GPUs,” https://docs.nvidia.com/datacenter/tesla/mig-user-guide/supported-gpus.html, accessed 2025-12-15.

[4] Amazon Web Services, “Accelerated computing: Amazon EC2 instance types,” https://aws.amazon.com/ec2/instance-types/accelerated-computing/, accessed 2025-12-15.

[5] Amazon Web Services, “Amazon EC2 G6 Instances,” https://aws.amazon.com/ec2/instance-types/g6/, accessed 2025-12-15.