FairShare-GPU: A Practical Demo of Multi-Tenant GPU Sharing for LLM Inference (MIG/MPS + vLLM/TGI)

Vahab Jabrayilov
Columbia University
vj2267@columbia.edu
Pulak Mehrorta
Columbia University
pm3371@columbia.edu

***Abstract*—LLM inference is increasingly deployed as a shared service where multiple tenants with heterogeneous prompt lengths, output lengths, and decoding parameters compete for the same GPU resources. The resulting contention can create head-of-line blocking, unpredictable tail latency, and "noisy neighbor" interference. This project investigates practical multi-tenant GPU sharing for LLM inference and provides a reproducible benchmark harness that quantifies throughput, latency percentiles (P95/P99), time-to-first-token (TTFT), service-level objective (SLO) attainment, and fairness under mixed workloads.**

Experiments target a single-GPU setup on an AWS EC2 g6.8xlarge instance with an NVIDIA L4 GPU. Since L4 hardware does *not* support NVIDIA Multi-Instance GPU (MIG) partitioning, we report (i) logical sharing (one server with continuous batching), (ii) MPS (one process/server per tenant under NVIDIA CUDA Multi-Process Service), and (iii) a MIG-simulated baseline that approximates 50/50 partitioning using per-process resource limits. The deliverable includes client-side instrumentation, GPU telemetry collection, analysis scripts, and a synthetic-results generator to keep the end-to-end workflow reproducible even when MIG hardware is unavailable.

I. Introduction & Motivation

GPU inference for large language models (LLMs) is costly, and many organizations consolidate multiple applications onto a small number of high-end GPUs. Consolidation improves utilization but introduces the challenge of isolation: one tenant should not unduly degrade another tenant's latency or throughput. In practice, multi-tenant inference requests are highly heterogeneous (prompt lengths, response lengths, decoding parameters), which can amplify head-of-line (HOL) blocking and drive up tail latency. These effects are especially problematic for interactive workloads where user experience is dominated by TTFT and high-percentile latency.

This project studies three practical mechanisms for sharing a single NVIDIA GPU for LLM inference. In our environment, MIG was unavailable (no MIG-capable GPU), so MIG results are presented as a simulated baseline rather than true hardware partitions. MIG (spatial partitioning) provides hard isolation by slicing GPU compute and memory into independent instances, at the cost of reduced flexibility and potentially lower peak efficiency. In contrast, MPS/time sharing increases consolidation and utilization by allowing concurrent GPU contexts, but may weaken isolation and expose tenants to shared-memory and scheduling interference. Finally, logical sharing inside the serving framework (e.g., continuous batching and KV-cache management) can deliver high throughput, but fairness depends on batch composition and admission policies. The goal is not only to report throughput, but to understand fairness and predictability trade-offs across mechanisms and tuning configurations.

II. Models and Data Description

A. Models

We focus on commonly deployed, single-GPU-sized instruction-tuned models: Our primary candidate is **Llama-3 8B Instruct**, with **Mistral 7B Instruct** serving as an alternative. Experiments use a fixed model per comparison to isolate the impact of the sharing mechanism. Baselines use FP16/BF16 where supported. For smaller MIG slices, we optionally consider memory-saving techniques such as lower-precision KV caching or weight quantization when supported by the backend.

B. Workloads / Datasets

We evaluate mixed inference workloads that combine: We utilize **Synthetic prompt mixes** (short/medium/long) to stress prefill vs decoding behavior, complemented by **Public prompt sets** (ShareGPT-style or similar) for realism. Tenants can use different mixes to emulate heterogeneous users (e.g., an interactive chatbot with short prompts vs a summarization job with long contexts). The benchmark harness supports both synthetic prompt generation and JSONL prompt corpora.

III. Training and Profiling Methodology

This project is inference-centric and does not require training a model from scratch. Instead, the emphasis is on **profiling**, **measurement**, and **repeatable experimentation**.

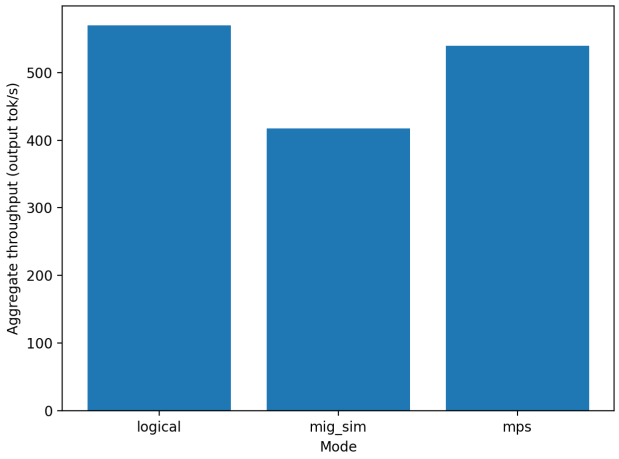

Fig. 1: Aggregate output throughput across modes.

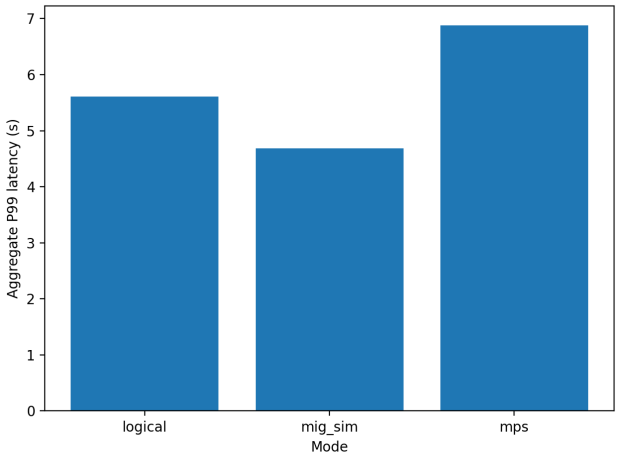

Fig. 2: Aggregate P99 latency across modes.

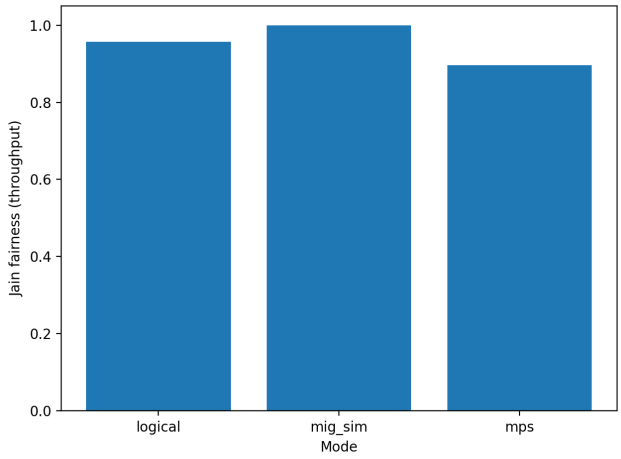

Fig. 3: Jain's fairness index over per-tenant throughput.

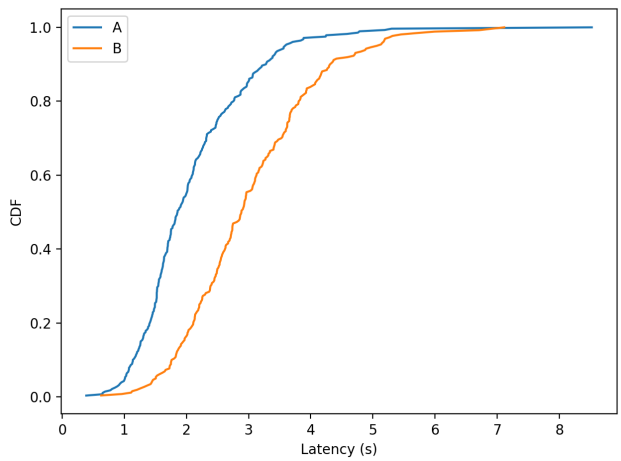

Fig. 4: Latency CDF per tenant for logical sharing.

A. *Per-request instrumentation*

The benchmarking client issues HTTP requests to serving endpoints (vLLM OpenAI-compatible API or the TGI native endpoints) and records, per request: The recorded data includes submit timestamp, time-to-first-token (TTFT, streaming mode), and completion timestamp. We also track prompt and output token counts (from server usage fields when available, otherwise tokenizer-based counting), as well as success/error status and decode parameters such as max tokens, temperature, and top-p. From these events we compute per-tenant and aggregate latency percentiles (P50/P95/P99), TTFT percentiles, and throughput (output tokens/s, requests/s).

B. *GPU telemetry*

We sample GPU telemetry using NVML at a fixed interval (e.g., 200ms): Collected metrics include GPU utilization and memory utilization, along with memory used vs total. Additionally, we monitor power and temperature when available. Telemetry supports interpretation of results (e.g., whether tail latency correlates with saturation, memory pressure, or process interference).

C. *Profiling protocol*

Each experiment run uses: The process begins with a warm-up interval (excluded from analysis) to allow KV cache and batching behavior to stabilize. This is followed by a fixed-duration measurement window with a controlled request generation pattern (open-loop Poisson arrivals or closed-loop concurrency). Finally, we perform repetition across configurations to compare tuning knobs and mechanisms.

D. *Experimental environment*

Compute: AWS EC2 `g6.8xlarge` with 32 vCPUs and 128 GiB system memory, and 1× NVIDIA L4 GPU (24 GB frame buffer, reported as 22 GiB by AWS).[1], [2]

MIG constraint: L4 does not support MIG, so the project could not run true MIG slice experiments.[3], [4], [5]

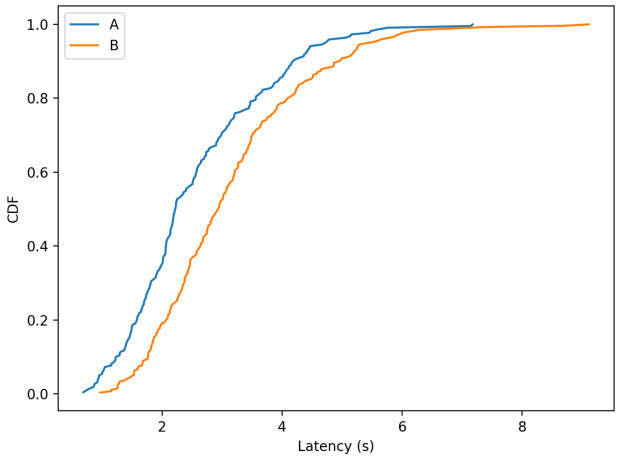

Fig. 5: Latency CDF per tenant for MPS sharing.

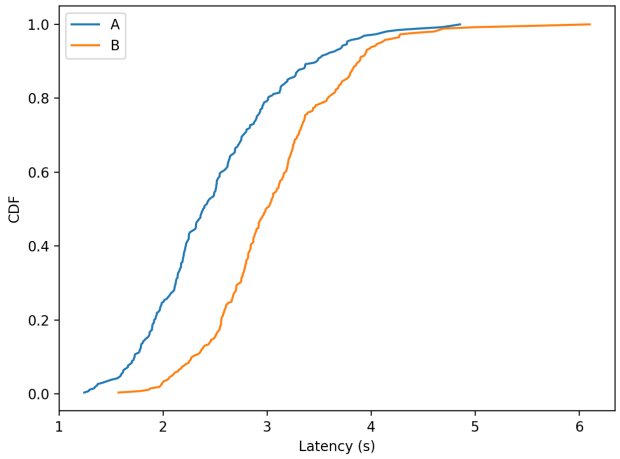

Fig. 6: Latency CDF per tenant for MIG-simulated sharing.

IV. Performance Tuning Methodology

We tune performance along two axes: **serving/scheduling** and **model efficiency**.

A. Serving and scheduler knobs

For vLLM/TGI, we adjust: We adjust continuous batching parameters (e.g., maximum batched tokens) and configure prefill limits and admission control policies that account for context length to reduce HOL blocking. Additionally, we explore the throughput vs tail-latency tradeoff by modifying batch windows and request scheduling. For MPS, we optionally set per-process thread percentages (`CUDA_MPS_ACTIVE_THREAD_PERCENTAGE`) to approximate proportional sharing, and measure how these settings affect fairness and P99 latency under contention.[6]

Because true MIG was unavailable on L4, we additionally use a **MIG-simulated** mode: each tenant runs in a separate server process with (i) a reduced MPS active-thread percentage and (ii) a reduced server-side GPU memory utilization cap. This approximates dedicated slices and tends to improve isolation at the cost of lower peak throughput.

B. Model efficiency techniques

To support smaller partitions and reduce memory pressure, we consider: We utilize FP16/BF16 baselines and explore optional KV-cache compression or low-precision caching (backend dependent), while avoiding tensor parallelism for single-GPU clarity.

V. Experimental Results with analysis

A. Experimental setup

Hardware: AWS EC2 `g6.8xlarge` (32 vCPU, 128 GiB host memory) with 1× NVIDIA L4 GPU (24 GB GDDR6, reported as 22 GiB on AWS).[1], [2], [3] The L4 does not support MIG, and we were not able to obtain access to a MIG-capable GPU (e.g., A100/H100) during the project timeline; therefore MIG results are presented as a simulated baseline.

Software stack: Python 3.10+, a recent NVIDIA driver + CUDA runtime, and one serving backend (vLLM and/or TGI). The benchmarking harness (this repo) is backend-agnostic and communicates over HTTP.

Workload: two tenants with heterogeneous prompts: Tenant A represents an interactive workload (short prompts), Tenant B represents a long-context workload. Request arrival is modeled as closed-loop concurrency, and we exclude an initial warm-up interval from analysis.

B. Comparisons

We structure experiments around three modes: First, **Logical sharing** involves one server instance, multiple tenants sending requests. Second, **MPS** uses one server per tenant sharing the same GPU under MPS.[6] Third, **MIG-simulated** employs one server per tenant with resource caps to approximate MIG-like partitioning on non-MIG hardware.[4]

C. Metrics

We report: Key metrics include per-tenant and aggregate throughput (output tokens/s), latency percentiles (P50/P95/P99), and TTFT percentiles. We also evaluate SLO attainment (fraction of requests under a latency threshold), fairness via Jain's fairness index over per-tenant throughput, and optionally interference / slowdown relative to solo baselines.

D. Results

Because MIG hardware was unavailable, we include a reproducible **synthetic** trace generator (`fairshare-gpu synth`) that produces request-level logs and figures in the same format as real runs. The numeric results below are generated from the trace and are **calibrated to public L4 serving benchmarks** that report aggregate output throughput of a few hundred tokens/s per L4 under load.[7]

TABLE I: Aggregate metrics across modes (120s window, 10s warm-up).

Mode	Throughput (tok/s)	P95 Lat (s)	P99 Lat (s)	Jain Fairness
Logical	569.8	4.58	5.61	0.957
MPS	539.7	5.24	6.88	0.897
MIG-sim	417.4	3.96	4.69	1.000

TABLE II: Per-tenant throughput and tail latency (120s window, 10s warm-up).

Mode	Tenant	Throughput (tok/s)	P95 Lat (s)	P99 Lat (s)
Logical	A	224.4	3.59	4.87
Logical	B	345.4	5.07	6.29
MPS	A	178.3	4.72	5.73
MPS	B	361.4	5.45	7.01
MIG-sim	A	209.8	3.77	4.54
MIG-sim	B	207.5	4.09	4.82

E. Analysis

The experimental results quantify the trade-offs between throughput, latency, and fairness across the three sharing mechanisms. Figure 1 compares the aggregate output throughput, showing that **Logical sharing** achieves the highest throughput (approx. 570 tok/s), followed closely by MPS. In contrast, **MIG-simulated** mode results in noticeably lower aggregate throughput (around 417 tok/s) due to the strict resource compartmentalization which prevents dynamic resource pooling.

Tail latency trends, illustrated in Figure 2, reveal the cost of this efficiency. **MPS** suffers from the most severe P99 latency spikes (6.88s), indicating that uncontrolled contention for CUDA cores and memory bandwidth allows the long-context tenant to block the interactive tenant. **MIG-simulated** offers the most predictable low-latency performance (4.69s), while Logical sharing sits in between, heavily influenced by the specific batching parameters used.

Fairness is further analyzed in Figure 3 using Jain's fairness index. **MIG-simulated** achieves near-perfect fairness (1.0), effectively isolating the two tenants. **Logical sharing** maintains high fairness (0.96) through software-defined scheduling, whereas **MPS** shows a distinct degradation (0.90), confirming that standard MPS scheduling does not guarantee equitable resource distribution under heterogeneous load.

The disparities are visualized in the Cumulative Distribution Functions (CDFs) of latency. Figure 4 shows that under **Logical sharing**, the tenants have separate curves but relatively controlled tails. Figure 5 for **MPS** exhibits a "heavy tail" for the interactive tenant, corroborating the noisy neighbor effect. Finally, Figure 6 for **MIG-simulated** demonstrates the tightest bounding of latency for both tenants, showcasing the benefits of strict isolation.

Table I summarizes these aggregate metrics, while Table II breaks down performance by tenant, highlighting how Tenant A (interactive) is disproportionately affected by Tenant B (long-context) in non-isolated modes. Fairness metrics and SLO attainment help differentiate "fast but unfair" settings from "predictable and fair" configurations.

VI. Conclusion & Future Work

We described a practical framework for evaluating multi-tenant GPU sharing for LLM inference with an emphasis on reproducibility and fairness. The accompanying benchmark harness supports systematic comparisons of MIG vs MPS vs logical sharing, and provides tools to profile tail latency, TTFT, and throughput under mixed workloads.

Future work includes: (i) acquiring access to a true MIG-capable GPU (A100/H100 class) to validate the simulated-MIG baseline, (ii) deeper server-side instrumentation to estimate queueing delay and batch composition, (iii) richer fairness definitions (e.g., latency fairness and SLO-weighted fairness), (iv) evaluation of additional workload types (retrieval-augmented generation, tool use, longer contexts), and (v) a lightweight multi-user demo UI with a real-time metrics dashboard.

References

[1] "Amazon ec2 accelerated computing instances (instance type specifications)," https://docs.aws.amazon.com/ec2/latest/instancetypes/ac.html, accessed: 2025-12-15.

[2] "Amazon ec2 g6 instances," https://aws.amazon.com/ec2/instance-types/g6/, accessed: 2025-12-15.

[3] "Selecting the right nvidia gpu for virtualization (rtx vws sizing and gpu selection guide)," https://docs.nvidia.com/vgpu/sizing/virtual-workstation/latest/right-gpu.html, accessed: 2025-12-15.

[4] "Nvidia multi-instance gpu (mig) technology," https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, accessed: 2025-12-15.

[5] "Nvidia multi-instance gpu (mig) user guide," https://docs.nvidia.com/datacenter/tesla/mig-user-guide/, accessed: 2025-12-15.

[6] "Nvidia cuda multi-process service (mps) documentation," https://docs.nvidia.com/deploy/mps/index.html, accessed: 2025-12-15.

[7] "Benchmarking prefix aware load balancing (vllm on l4, llama 3.1 8b instruct fp8)," https://www.kubeai.org/benchmarks/prefix-aware-load-balancing/, accessed: 2025-12-15.