Characterizing CPU-Induced Slowdowns in Multi-GPU LLM Inference

Euijun Chung, Yuxiao Jia, Aaron Jezghani, Hyesoon Kim
Georgia Institute of Technology
{*euijun, yjia305, ajezghani3*}*@gatech.edu, hyesoon@cc.gatech.edu*

***Abstract*—Large-scale machine learning workloads increasingly rely on multi-GPU systems, yet their performance is often limited by an overlooked component: the CPU. Through a detailed study of modern large language model (LLM) serving workloads, we find that multi-GPU performance often degrades not because GPUs are saturated, but because CPUs fail to keep them busy. Under limited CPU allocations, systems exhibit symptoms such as delayed kernel launch, stalled communication, and increased tokenization latency, leading to severe GPU underutilization even when ample GPU resources are available. The problem becomes more severe in agentic LLM serving, where long accumulated contexts increase CPU-side tokenization work while high prefix-cache reuse across multi-turn interactions reduces GPU-side prefill work. These bottlenecks persist even in serving stacks that employ process-level separation and modern GPU-side optimizations such as CUDA Graphs. Since CPU cores cost orders of magnitude less than GPUs, provisioning additional cores is a highly cost-effective mitigation. Under moderate serving load, we observe that CPU-starved configurations frequently time out, while providing adequate CPU resources restores responsiveness and reduces time-to-first-token (TTFT) latency by 1.47–7.11× across configurations, all without requiring additional GPUs.**

***Index Terms*—Workload characterization, LLM serving, CPU bottleneck, Multi-GPU systems**

I. Introduction

Modern machine learning (ML) workloads require massive computational power, driving the widespread adoption of multi-GPU servers, such as the DGX H100 and DGX B200 [49], [50]. GPU servers are expensive to purchase and operate, requiring substantial electricity and cooling infrastructure. To ensure efficient use of valuable resources, organizations operate them in a shared, multi-tenant manner, allowing multiple users to work on their own allocated virtual instances. Through resource schedulers such as Slurm or cloud platforms (e.g., AWS and Azure), users request fixed, manually specified allocations (e.g., 4 GPUs and 8 CPU cores) at a given cost per unit time [3], [68], [78]. While this approach improves GPU utilization across many users, it creates a critical blind spot: the CPU is often underprovisioned relative to GPUs. In this paper, we reveal why this imbalance can silently turn the CPU into a first-order bottleneck in multi-GPU large language model (LLM) inference, even when the GPUs themselves have ample compute capacity.

In multi-GPU ML workloads, CPUs perform numerous critical tasks, and systems with limited CPU resources can experience severe slowdowns when CPUs are oversubscribed. The CPU's tasks include data I/O and processing, inter-GPU synchronization, request scheduling, and kernel launches [38], [39], [48]. A significant portion of a CPU's job involves feeding sufficient data to GPUs, i.e., retrieving data from host memory and performing preprocessing steps such as tokenization for language models [27], or image decoding and augmentation for vision models [34], [52], [71], [80]. Moreover, CPUs are responsible for coordinating computation and communication across multiple GPUs by issuing kernel launches, managing stream dependencies, and orchestrating synchronization [5], [36], [79].

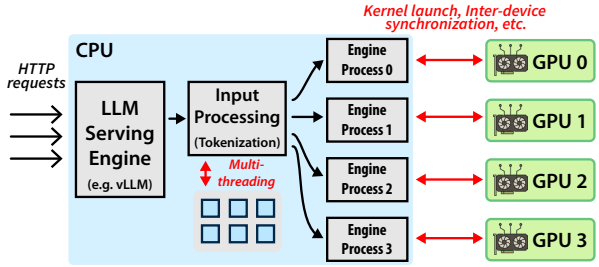

Fig. 1. CPU tasks in multi-GPU LLM serving: input processing, kernel launches, and synchronization.

However, once such processes contend for limited CPU cores, oversubscription increases host-side latency and slows the launch of both compute and communication kernels. This behavior is especially pronounced during LLM serving with multiple incoming requests, where a high CPU load from these requests can substantially increase kernel launch latency from microseconds to milliseconds, thereby degrading overall performance and risking service-level objective (SLO) violations [22], [90].

Collective communication libraries, such as NCCL [26], [48], can lead to additional slowdowns in scenarios with limited CPU resources, as their collectives require all participating GPUs to synchronize. If the CPU is late to dispatch on any rank, the entire device group stalls. While CUDA Graphs [47] amortize launch overhead for static kernel sequences, they cannot capture dynamic scheduling decisions required at every decode step of LLM inference [14], leaving CPU-side bottlenecks as a persistent limiting factor.

The CPU bottleneck becomes even more severe in agentic and multi-turn LLM serving. As contexts grow across turns, high prefix-cache reuse reduces GPU-side prefill work, while the CPU must still tokenize the full accumulated context at every turn. This imbalance increases the relative importance

of CPU-side work and makes the system more sensitive to CPU scarcity.

The goal of this work is to challenge a common misconception among multi-GPU system users: the assumption that GPU performance alone determines the efficiency of modern ML workloads. Prior studies have shown that host-side orchestration and CPU single-thread performance can affect LLM inference latency in single-GPU or CPU-GPU coupled settings [73], [74], and practitioners have noted that CPU performance matters in multi-GPU systems [57], [72], [90]. However, it remains unclear why limited CPU resources can cause severe slowdowns, specifically in multi-GPU LLM serving, where the symptoms often appear as low GPU utilization, time-to-first-token (TTFT) spikes, or request timeouts.

This paper explains why CPU scarcity becomes a performance bottleneck in multi-GPU LLM serving by identifying three critical host-side paths. First, tokenization threads compete with latency-sensitive LLM engine processes for CPU cores, delaying scheduling and kernel dispatch. Second, late CPU-side kernel launches on any GPU rank create stragglers at collective synchronization points, forcing other GPUs to wait. Third, the shared-memory broadcast queue used to deliver scheduling metadata from the engine to GPU workers can itself become a CPU-side delay on the critical path. Together, these mechanisms explain how CPU-side delays propagate into GPU idle time and end-to-end latency, even in optimized serving stacks with CUDA Graphs, `torch.compile`, and process-level isolation.

We evaluate these effects under varying CPU resource configurations in LLM inference and online serving scenarios in Section IV, then analyze the underlying synchronization and shared-memory bottlenecks in Section V. Section VI discusses deployment implications, including a case study on a high-performance computing (HPC) cluster and cloud CPU economics. Section VII reviews related work, and Section VIII concludes.

Our contributions are as follows:

- **Characterization.** In long-context LLM serving with high prefix-cache reuse, we show that CPU scarcity can substantially increase TTFT. Across dense and MoE models on 4–8 GPUs, CPU-starved configurations experience up to 7.11× higher TTFT and engine timeouts compared with CPU-abundant configurations.
- **Mechanisms.** We identify two mechanisms by which CPU contention causes end-to-end slowdowns: delayed kernel dispatch interacting with collective synchronization barriers, which inflates collective time by 3.47×, and tail stalls in the engine-to-worker broadcast path that delay GPU workers.
- **Deployment relevance.** We analyze 4.65 million real-world cluster allocation records to show that CPU under-provisioning is common in practice, and demonstrate that increasing CPU allocation can be a practical mitigation given its relatively low marginal cost compared with GPU resources.

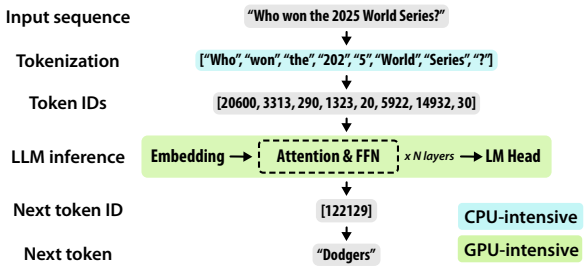

Fig. 2. LLM inference pipeline: CPU-intensive tokenization followed by GPU-intensive forward pass of the LLM.

II. Background & Motivation

A. CPU's Role in LLM Serving

In modern LLM serving systems, the CPU is responsible for several latency-sensitive operations that directly influence end-to-end performance. As shown in Figure 1, the CPU's job in LLM inference includes: ① **processing and tokenizing input text**, ② **handling HTTP requests and scheduling them**, and ③ **issuing GPU kernel launches** to sustain steady execution on the accelerators. Although GPUs perform the bulk of the model's numerical computation, the CPU remains responsible for coordinating each step of computation, and these input processing and control-plane tasks can become the dominant bottleneck in multi-GPU inference.

① **Tokenization** adds substantial CPU work during LLM inference. As shown in Figure 2, tokenization converts raw text strings into integer token IDs that the model can process using subword segmentation algorithms such as byte pair encoding (BPE) [64] or SentencePiece [35]. This process occurs at the beginning of every inference request for prompt processing. Tokenization consumes substantial CPU cycles, particularly for long prompts or batched requests, where preprocessing costs scale linearly with the input size. Because tokenization is a prerequisite to any LLM computation, it lies on the critical path of each request and adds directly to end-to-end latency unless overlapped with the GPU-side computation of other requests.

Widely used tokenizer implementations employ multi-processing or multithreading to spread the overhead of text processing across multiple CPU cores. In particular, the HuggingFace Tokenizers library [27] enables its Rust-based tokenizer to spawn multiple parallel threads, with `TOKENIZERS_PARALLELISM=true` as the default setting.

② **HTTP server request handling** also adds CPU load through connection management, request parsing, and batched scheduling. This overhead is typically much smaller than the tokenization cost, so our analysis focuses primarily on tokenization.

③ **Kernel launch overhead** represents another source of CPU work. The CPU must schedule and dispatch CUDA kernels to sustain model execution, intervening at least once per generated token: each token requires host-side processing, such as sampling, end-of-sequence (EOS) detection, or agentic tool-call dispatch, before the next step can be launched. If the

TABLE I
LIST OF CPU-GPU HETEROGENEOUS SYSTEM SETUPS USED IN THIS PAPER'S EVALUATION.

System (GPU)	Architecture (Compute Capability)	CPU Model	#CPU Cores	#GPUs per Node	Inter-GPU Interconnect
H100	Hopper (9.0)	Intel Xeon Platinum 8462Y+	64	8	NVLink 4.0 (900 GB/s bidirectional)
H200	Hopper (9.0)	Intel Xeon Platinum 8562Y+	64	8	NVLink 4.0 (900 GB/s bidirectional)
RTX Pro 6000	Blackwell (12.0)	Dual Intel Xeon 6737P	64	8	No NVLink (PCIe 5.0, 64 GB/s)

host thread responsible for kernel dispatch is delayed by OS scheduling or CPU contention, the GPU idles until the launch completes, slowing the entire pipeline.

CUDA Graphs [47] amortize most repeated launch calls by capturing and replaying complete computation sequences. However, they cannot capture operations that remain dynamic, including stream synchronizations, library kernels outside the graph, and the per-token host-side processing above. CUDA graph launch must still be issued by the CPU at every decode step [14], [32].

To ensure that GPUs receive kernels on time, multi-GPU ML and LLM libraries such as `torch.distributed` [38] rely on multiprocessing to allow each process to launch kernels independently and provide sufficient concurrency for high-throughput GPU dispatch. As a result, multi-GPU inference frameworks, including vLLM [36], DeepSpeed [62], and HuggingFace Accelerate [21] allocate at least one process per GPU, as also depicted in Figure 1. The key takeaway is that modern multi-GPU frameworks should assign at least one process per GPU to enable efficient kernel dispatch and consistently keep the accelerators fed.

Taken together, when CPU cores are underprovisioned, data-processing tasks compete with the LLM engine processes for cores, and the entire GPU workload pipeline can be substantially delayed. In particular, under concurrent LLM requests, multiple requests can keep the tokenizer heavily occupied and contend with the kernel-launching processes, introducing delays even on systems with seemingly sufficient CPU resources. This competition causes cascading slowdowns in kernel dispatch and request scheduling, ultimately degrading the end-to-end performance. We present experiments in Section IV that show how the CPU can cause end-to-end slowdowns, along with an analysis of latency and CPU and GPU utilization. Additionally, Section V-A examines the effect of CPU oversubscription in more detail, visualizing cases in which CPU-side resource contention delays kernel calls and results in GPU underutilization.

B. Agentic LLM Workloads and Prefix Caching

With the rise of agentic AI, LLM inference increasingly runs inside multi-turn loops that interleave reasoning with tool invocations [32], [61], [84], [89]. Agentic and multi-turn workloads resubmit the entire accumulated context at every turn: the model re-reads the conversation history plus a short appended delta, such as a tool result, before producing its next action [37], [40], [87]. Trace studies show that while contexts accumulate to tens or hundreds of thousands of tokens over a session, each turn appends only a few hundred new tokens [13], [59]. Serving engines therefore rely on prefix caching to reuse the key-value (KV) cache of previously processed tokens; in such sessions, over 95% of each turn's input tokens are served from the cache, reducing per-turn GPU prefill to the small uncached delta [19], [20], [81], [88], [95].

Prefix caching, however, does not reduce the CPU-side cost of a turn. In mainstream engines such as vLLM [36] and SGLang [92], a request must be fully tokenized before the cache can be queried, so each turn re-tokenizes the entire accumulated context. This cost grows with session length [65], while the GPU prefill remains bounded by the small per-turn delta. As cache hit rates rise, the CPU accounts for an increasing share of TTFT, and this shift is the regime our work characterizes.

III. MULTI-GPU SYSTEM EVALUATION SETUP

Table I lists the H100, H200, and RTX Pro 6000 systems used in our evaluation. We disable simultaneous multithreading (SMT) to isolate effects from physical cores. Since cloud platforms typically expose vCPUs, our measurements provide a conservative baseline for CPU provisioning. We run our evaluations on the listed machines with restricted (affinity-limited) CPU allocations to examine how multi-GPU setups in multi-tenant clusters can lead to unexpected performance degradation due to the limited CPU resources. For example, on a server equipped with four H100 GPUs, we may create a constrained execution environment with only 16 CPU cores to emulate a realistic cloud or cluster.

All LLM experiments use vLLM v0.21.0 with the V1 engine, which separates API serving/tokenization from EngineCore scheduling and GPU worker execution across processes. Default optimizations, including CUDA Graphs, prefix caching, `torch.compile`, asynchronous scheduling, and custom `AllReduce` kernels, remain enabled. Thus, the observed bottlenecks persist in an optimized serving stack rather than an intentionally unoptimized baseline.

Moreover, we use a fixed CPU & GPU resource allocation per job, reflecting common practice in cluster schedulers such as Slurm [68] and in cloud service offerings where users receive a predetermined number of CPU cores per GPU [3]. While such isolation supports fair sharing across users, it also introduces the risk of mismatched CPU and GPU allocations, as these allocations are typically chosen manually or determined by simple heuristics (e.g., assigning 4 CPU cores per GPU).

IV. THE CPU BOTTLENECK IN LLM INFERENCE

In this section, we analyze how CPU-side tasks during LLM inference and serving, such as tokenization, runtime orchestration, and communication coordination, can become

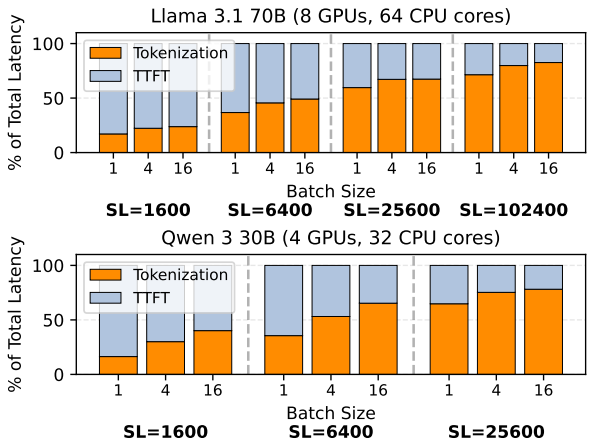

Fig. 3. Tokenization vs. time-to-first-token (TTFT) latency across batch sizes and sequence lengths (SL). Llama on 8×H200 (64 CPU cores). Qwen on 4×H200 (32 CPU cores).

dominant bottlenecks when multiple jobs or requests execute concurrently. We show that contention for limited CPU resources not only delays host-side processing but also slows GPU execution, resulting in increased per-token latency and reduced overall throughput.

A. Tokenization Latency Evaluation in LLM Inference

Figure 3 visualizes how much of the end-to-end LLM prefill latency is attributable to tokenization. We use vLLM [36] to systematically quantify the relative cost of CPU-side tokenization compared to GPU-side model execution; similar CPU-side bottlenecks are expected in other frameworks such as SGLang [92] with a multi-process architecture. We evaluate two models with distinct sparsity characteristics: a dense model (Llama 3.1 70B) on an 8×H200 system with 64 CPU cores, and a Mixture-of-Experts (MoE) model (Qwen3 30B, 3B active parameters) on a 4×H200 system with 32 CPU cores. Both use tensor parallelism (TP) [66] and HuggingFace Tokenizers [27] as integrated in vLLM. We measure TTFT as the wall-clock duration of `llm.generate()` function with pre-tokenized inputs (engine-internal tokenization bypassed), averaged over five iterations after three warmup runs. We measure tokenization latency separately via `tokenizer.encode()` calls.

For each trial, we vary batch size $\{1, 4, 16\}$ and input sequence length $\{1600, 6400, 25600, 102400\}$, ranging from a single interactive request to a moderately batched, long-context multi-tenant load [59]. To emulate steady-state agentic turns, each request appends a fresh 256-token tool-call result to a context with a warmed prefix cache, reflecting observed per-turn appends of a few hundred tokens [13].

Results. Tokenization accounts for up to roughly 80% of TTFT in long-sequence, large-batch configurations across both dense (Llama) and MoE (Qwen) architectures. The fraction grows with sequence length by construction of this regime. Tokenization processes the full accumulated context, whereas GPU prefill computes only the fixed 256-token delta; the CPU-side cost therefore scales with sequence length while the GPU-side cost stays nearly constant. As sessions approach million-token contexts, per-turn tokenization alone can require seconds of CPU time [76]. While the relative impact of tokenization decreases once decode is taken into account, agentic sessions re-tokenize the full context at every turn, so tokenization returns to the critical path at each turn's TTFT.

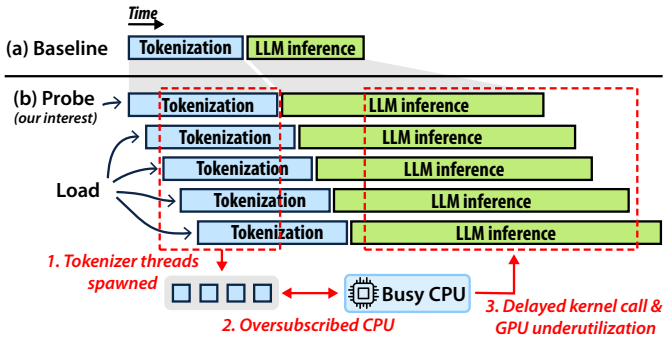

Fig. 4. (a) Baseline LLM inference. (b) Probe under load: (1) tokenizer spawns threads, (2) CPU saturates, and (3) delayed kernel launches underutilize GPUs.

B. Impact of Tokenization Load in LLM Serving

Next, we evaluate how a busy tokenizer can slow down the LLM engine. Our experiment shows that under heavy load, multiple tokenizer threads can compete with the LLM engine processes for CPU resources, which leads to frequent context switches, delayed kernel launches, and reduced GPU utilization. Such a slowdown substantially degrades latency as communication kernels rely on barrier-based synchronization, as discussed further in Section V-A.

Evaluation methodology. We design an experiment where multiple concurrent requests are sent to the LLM server, and we measure how this background load affects the latency of a single target request. We refer to the measured request as the *probe*, and we generate additional periodic requests, termed *load*, that impose a controlled CPU load defined by a specified requests per second (RPS) rate. Note that both request types are served by a single LLM engine and batched together through continuous batching, sharing the same model instance without any partitioning of CPU or GPU resources [86].

The overall setup is illustrated in Figure 4, which shows how a four-request load after the probe keeps the tokenizer busy, thereby delaying both tokenization and inference due to delayed kernel calls and GPU underutilization. We implement this experiment using `vllm serve` with the V1 async engine described in Section III, where user queries arrive asynchronously over HTTP. Despite V1's process-level separation of tokenization, scheduling, and GPU execution, CPU contention persists because all processes compete for the same limited pool of CPU cores. In this experiment, we quantify CPU contention by measuring the latency of the probe under load while varying the number of CPU cores. This approach distinguishes CPU- versus GPU-limited regimes in multi-GPU LLM serving.

LLM serving system setup. We evaluate three models spanning sizes and sparsity regimes: Llama 3.1 8B (dense),

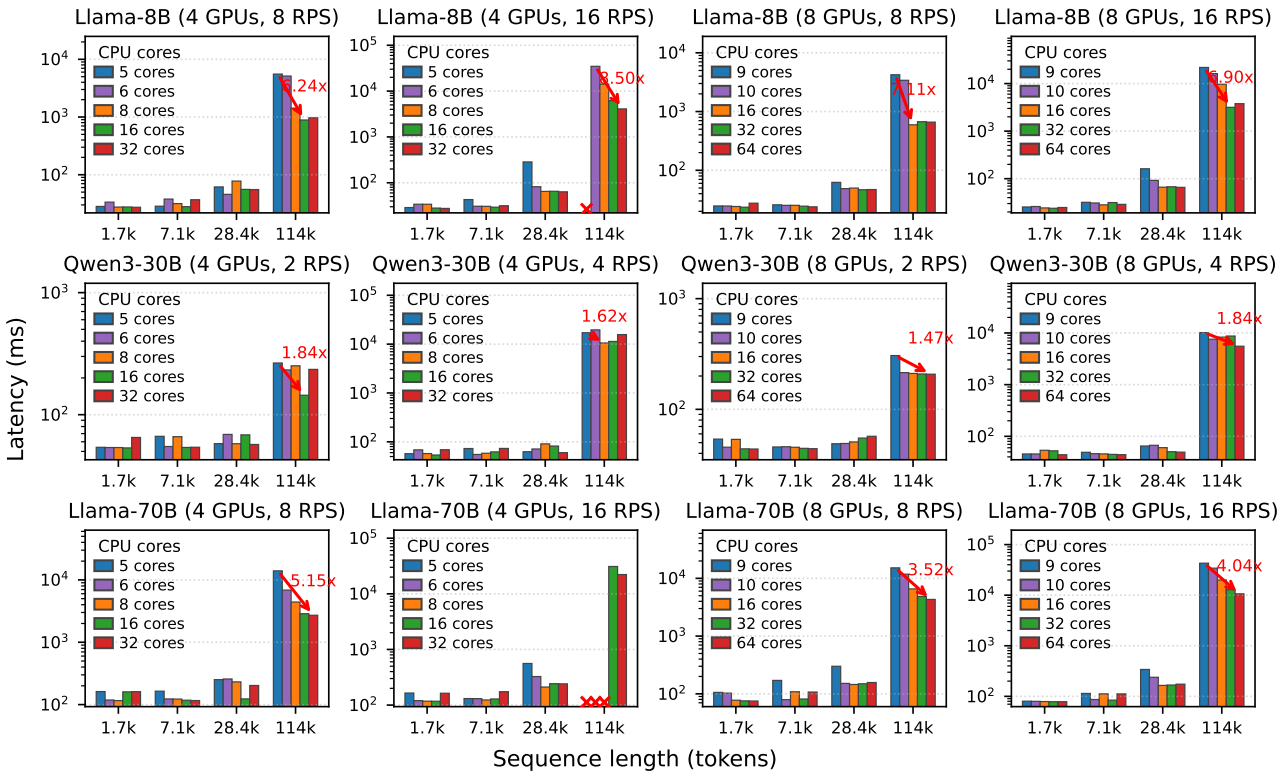

Fig. 5. Probe TTFT across models on 4- and 8-GPU configurations: Llama 3.1 8B (H100, 8/16 requests per second (RPS)), Qwen3 30B (H100, 2/4 RPS), Llama 3.1 70B (H100, 8/16 RPS). Red × marks 200-second timeouts; red arrows show speedup from smallest to best CPU allocation.

Qwen3 30B (MoE), and Llama 3.1 70B (dense), each in 4-GPU (TP=4) and 8-GPU (TP=8) configurations. All models run in bfloat16 (bf16) precision. CPU resources (#CPU) are provisioned at five levels: (#GPUs + 1) cores (least-CPU case), (#GPUs + 2) cores (matching vLLM V1's concurrent-process count), 2×#GPUs, 4×#GPUs, and 8×#GPUs (CPU-abundant cases). As described in Section III, vLLM V1 requires at least (#GPUs + 2) concurrent processes; our least-CPU configuration of (#GPUs + 1) cores is therefore already below this minimum, guaranteeing that at least one process must time-share a core via OS scheduling. In practice, allocating exactly #GPUs cores causes the system to operate prohibitively slowly, so we use (#GPUs + 1) as the bare-minimum functional configuration. For every configuration, we report the client-side TTFT under the assumption of a prefill-decode-disaggregated deployment for the prefill stage [56], [59], [94].

The load's RPS values differ across models: Llama uses 8 and 16, while Qwen3 30B uses 2 and 4 because the higher rates time out across all cells for the sparse MoE Qwen3 30B. The load's sequence length ranges from 1.8k to 114k tokens, while the probe's sequence length is 2.8k tokens. There are slight differences in the exact token count between the Llama and Qwen models. We measure the average latency of four probes after dropping the first post-load sample as a warmup. We set the output length of each load request to one token to isolate long-context request processing from sustained decode load. Both parameters reflect realistic serving conditions [77], [82]: the request rates fall within observed production ranges (the tool-and-agent workload trace in Mooncake [59] averages 6.7 RPS, with bursts up to 23). The sequence lengths also represent moderate load for long-context serving, given that recent models such as Qwen3.5-Flash [70] and Nemotron 3 [4] support context windows of up to 1M tokens. Prefix caching is enabled, reflecting reuse from shared system prompts and accumulated tool-call history for the load. Our synthetic load reaches KV cache hit rates of 92.5%, 78.1%, and 95.3% for Llama 8B, Qwen 30B, and Llama 70B, respectively. However, each probe carries a unique prefix and is served entirely uncached, so its TTFT includes tokenization and a full prefill pass.

Latency results. Figure 5 summarizes the TTFT of probes under a load at a given RPS. Across the configurations, increasing #CPU can eliminate engine timeouts (marked in red ×) or reduce the probe's TTFT, confirming that the CPU bottleneck is not specific to a single system or a model. Removing CPU scarcity improves long-sequence TTFT by 1.47–7.11× when we go from (#GPUs + 1) cores to a CPU-abundant configuration.

This improvement stems from two effects: ① tokenizer threads in the API server face less contention and finish sooner, and ② host-side tasks across the EngineCore (batching, scheduling, and memory management) and GPU worker processes (kernel dispatch and data transfers) proceed without delay. Even at (#GPUs + 2) cores, where every engine process holds a dedicated core, the probe's TTFT remains 2.63× (geometric mean) above the CPU-abundant configuration: the

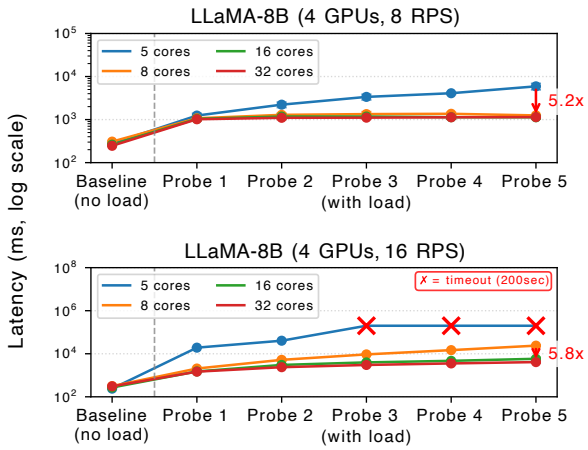

Fig. 6. Sequentially issued probes' TTFT under load (8/16 requests per second (RPS), 114k tokens); more CPU cores reduce the slowdown. Measured on Blackwell GPUs.

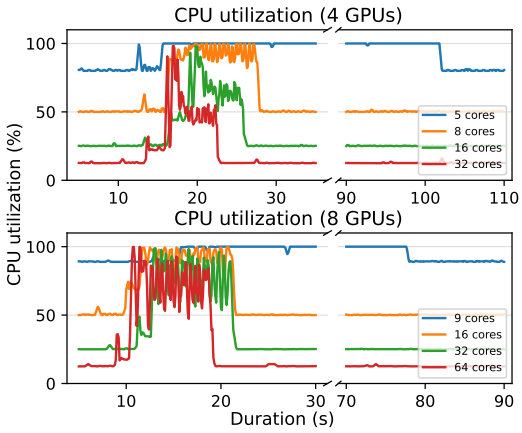

Fig. 7. CPU utilization (Llama 3.1 8B) across core allocations on 4-/8-GPU configurations; more cores shorten the saturation period.

tokenizer alone spawns more threads than the remaining cores, so contention persists until CPU cores well outnumber the engine's processes and threads.

With higher load RPS, fewer-core systems saturate earlier: context switching spikes, kernel launches serialize, and GPUs sit idle even when work is available. Larger CPU allocations tolerate much higher load before throughput collapses, so CPU headroom directly raises the sustainable request rate. However, at the largest 64-core allocation, we occasionally observe a slight latency rebound, likely from cross-socket non-uniform memory access (NUMA) effects and over-parallelization in the tokenizer's thread pool.

MoE models are more sensitive to CPU scarcity than dense models. In our load experiments, Qwen3 30B (MoE) reaches engine saturation at far lower rates (RPS=2–4) than Llama 3.1 70B (dense) on the same H100 system (RPS=8–16). Because an MoE model activates only a few experts per request, its per-step GPU time is shorter, so CPU-side work occupies a larger fraction than in a dense model [17]. This empirically corroborates the single-node host-bound analysis in TaxBreak [74], which reports that MoE inference is particularly sensitive to host CPU performance. Our multi-GPU concurrent-serving experiments extend that observation: under sustained load, the smaller per-step GPU usage of MoE amplifies CPU-side queueing and produces an earlier transition to the CPU-bound regime.

LLM engine timeouts. Some configurations fail to produce latency values, marked by red × in Figure 5. The failure follows a positive-feedback loop. Tokenization from the load saturates the CPU, delaying scheduling and kernel launches. Then, the resulting higher latency keeps requests active longer, so more arrivals accumulate in the engine queue and further amplify tokenization and scheduling contention. Eventually, the system enters an overloaded state where the request queue grows unbounded and the probe request times out at our 200-second limit [46]. The feedback is accelerated on MoE models: if concurrent requests activate different experts, the batch touches more expert weights per step, lengthening per-step GPU time and growing the queue at far lower request rates.

Latency per request. Figure 6 reports the TTFT latencies of the baseline and five probes under load of 8/16 RPS on Llama 3.1 8B with TP=4, sequence length (SL)=114k. Probes are issued sequentially, so as the load's requests accumulate in the serving engine, each subsequent probe's TTFT shows an increasing trend. In particular, scaling from 5 CPU cores (the least-CPU configuration) to 32 cores consistently yields over 5× lower TTFT.

CPU and GPU utilization. We sample CPU and GPU utilization at 100 ms intervals using `psutil` and `nvidia-smi`, under 8 RPS with 114k-token sequences. Traces are shown for Llama only, as Qwen3 30B behaves qualitatively similarly. Figure 7 shows that CPU-induced latency correlates more strongly with the duration of CPU saturation than with peak utilization. With 5 cores, the CPU stays near 100% for up to 100 seconds, and preprocessing tasks queue behind it. With 8–32 cores, utilization still spikes, but the spikes drain within seconds, and the GPU launch path remains supplied with work; the 8-GPU configuration behaves the same up to 64 cores. Figure 8 shows the consequence on the GPU side: during CPU saturation, the GPU sits idle, and with sufficient cores the same work finishes in a visibly shorter span. Note that `nvidia-smi` counts a GPU as busy even while kernels spin-wait at synchronization barriers, so these traces measure occupancy rather than useful work; Section V-B quantifies effective kernel execution directly.

Effect of further host-side optimizations. We argue that while host-side optimizations continue to emerge, the CPU bottleneck is still likely to cause slowdowns and engine timeouts. For instance, `fastokens` [9] is a recent tokenizer backend that outperforms HuggingFace Tokenizers on long contexts. To quantify the remaining CPU overhead, we repeat the serving experiment for Llama 3.1 70B (4 GPUs) with `fastokens` enabled. The faster tokenizer only raises the load threshold at which the engine collapses: the least-CPU configuration, which previously timed out at 16 RPS, now completes. However, its TTFT remains 2.13× higher than the

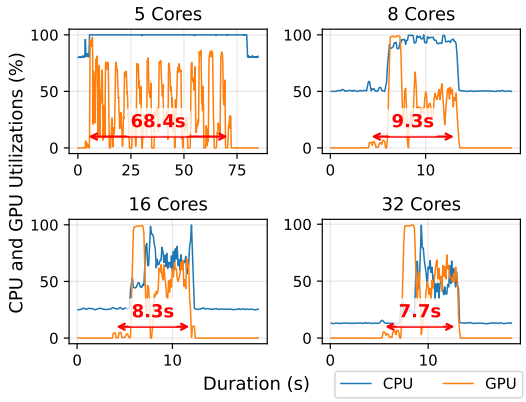

Fig. 8. CPU/GPU utilization (Llama 3.1 8B, 4-GPU) across CPU allocations: CPU saturation leaves GPUs idle.

CPU-abundant configuration, and the engine still times out at 28 RPS. Host-side optimizations thus shift the onset of CPU saturation to higher load but do not remove the CPU from the critical path.

Takeaways. In multi-request LLM serving pipelines, tokenization adds substantial pressure on the CPU, and this often causes end-to-end latency to be limited not by GPU throughput but by the CPU's ability to perform tokenization and kernel scheduling on time. Under concurrent load, these CPU bottlenecks delay kernel launches, stall preprocessing pipelines, and leave GPUs underutilized. The "probe under load" experiment shows that allocating sufficient CPU resources is crucial for achieving high serving performance and preventing engine timeouts, particularly under high-RPS or long-context workloads where tokenization dominates the total work.

V. Understanding the CPU Bottleneck in Multi-GPU Systems

This section analyzes how exactly CPU oversubscription and resource contention among multiple processes degrade multi-GPU performance by examining the CPU's role in kernel launches and GPU synchronization. We show that since communication kernels such as `AllReduce` synchronize all GPUs at a barrier, every GPU must busy-wait for the slowest rank, and this can stall the entire group when CPU resources are insufficient. Additionally, we find that contention on shared regions such as the Linux shared-memory segment increases launch-path latency when many processes compete for access.

A. Synchronization and CPU Oversubscription in Communication Kernels

To examine how CPU contention affects GPU utilization, we develop a microbenchmark with the `torch.distributed` framework [38] and profile it using the PyTorch Profiler [58]. The experiment isolates CPU-side kernel dispatch and GPU-side execution, allowing us to quantify the synchronization overheads induced by CPU oversubscription.

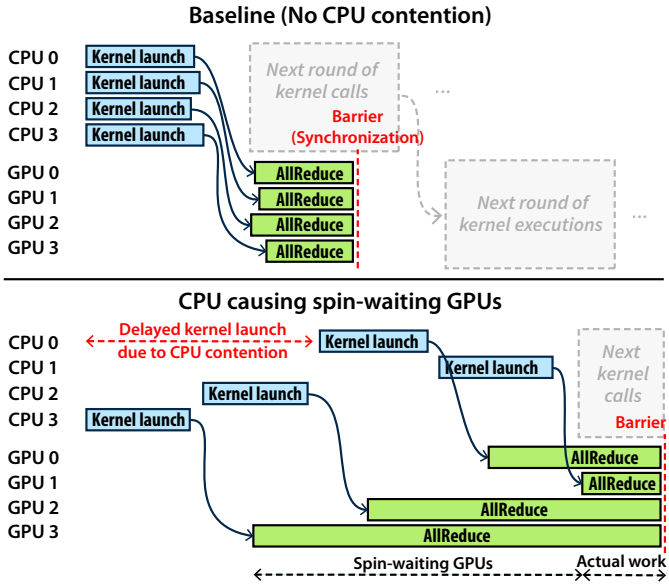

Fig. 9. Profiled timeline showing how contended CPU cores serialize kernel launches and underutilize GPUs. Gray boxes mark subsequent kernel calls.

Figure 9 illustrates the profiled result of communication kernels under extreme CPU contention, where multiple processes contend for a single CPU core. In this configuration, the CPU can launch only a single kernel at a time, forcing the GPU kernels to start sequentially rather than concurrently. Since NCCL collectives such as `AllReduce` introduce global synchronization barriers, faster GPUs must wait until every process has reached the barrier, creating idle busy-wait periods that waste compute cycles (highlighted in the figure by black dotted lines). This produces a *straggler* effect: in an 8-GPU collective, if a single CPU core is delayed by 1 ms due to OS scheduling, all other GPUs busy-wait for that duration, amplifying a small per-core delay into a cluster-wide stall.

This *straggler* phenomenon generalizes beyond collectives: any workload stage that involves CPU-side coordination, such as `DataLoader` workers, tokenization threads, or request-handling processes in serving engines, can trigger similar stalls when CPU cores are oversubscribed. Thus, provisioning CPU cores beyond the minimum process count is necessary to avoid these oversubscription-induced slowdowns. Analytical performance models have independently identified collective communication synchronization as a fundamental bottleneck in distributed LLM inference [11]; our measurements provide empirical confirmation of this on real hardware under CPU-constrained conditions.

B. Kernel-Level Analysis of CPU-Induced Slowdown

To confirm that the slowdown does not originate from GPU execution, we profile individual GPU kernels from the "probe under load" experiment, averaged across ranks. Figure 10(a) shows that per-kernel execution time of compute kernels is unaffected by #CPU: GEMM, attention, and elementwise kernels differ by less than 1% between 6 and 32 cores. In contrast, the mean duration of communication collectives

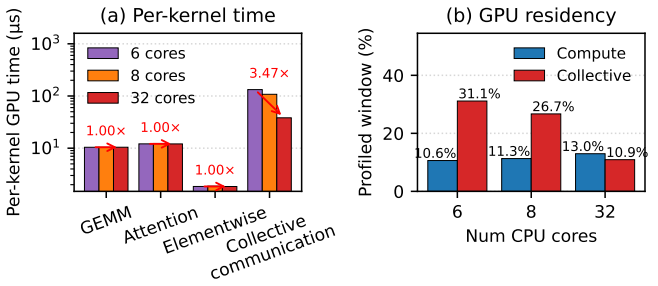

Fig. 10. Kernel-level profile under probe load (Llama 3.1 8B, TP=4, 16 RPS). (a) GPU time per kernel. (b) Fraction of kernel execution time relative to the profiled window of 300 engine iterations.

inflates by 3.47×, and the inflation is asymmetric across ranks. For instance, at 6 cores, the p99 latency for the collective is 1.3 s on rank 0 but 5.1–6.2 s on the remaining ranks, which wait at the barrier for the delayed peer. These results confirm that slower GPU compute execution is not the cause of the observed end-to-end slowdown [7].

Figure 10(b) decomposes GPU residency over the profiled window of 300 engine iterations. As CPU cores increase from 6 to 32, the time collectives spend spinning at synchronization barriers shrinks, and compute kernels take up a larger share of time within the window. Because a spin-waiting collective still counts as "busy" in coarse utilization counters such as `nvidia-smi`, this decomposition reveals what such counters conceal: at 6 cores, most of the reported GPU activity is barrier spinning rather than useful work.

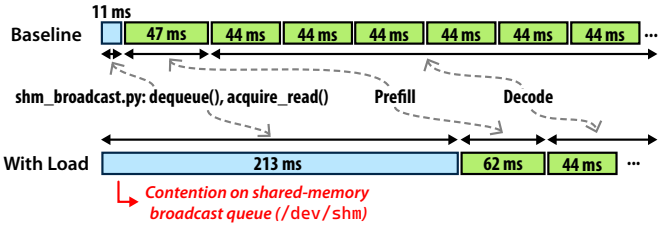

Fig. 11. Contended `dequeue()` on the shared-memory broadcast queue delays scheduling metadata to GPU workers (timing examples from Llama 3.1 8B).

C. Shared Memory Broadcast Contention

Modern LLM serving frameworks such as vLLM [36] and SGLang [92] share a multi-process architecture in which the scheduler broadcasts work to all workers. We show that this broadcast introduces a second straggler effect, this time on the CPU side. The broadcast follows a 1-writer-N-reader pattern, where N equals the number of GPUs. The writer (engine core) must verify that all N readers have consumed the previous message before writing new data, and each reader must confirm the writer has produced a new entry before consuming it. Although the usual implementation is lock-free (e.g., vLLM), both the writer and the readers busy-wait without yielding to sleep. Under CPU scarcity, the writer's busy-wait competes with reader processes for CPU time, delaying readers' flag updates and forcing the writer to spin longer, creating a cascading delay that amplifies as the

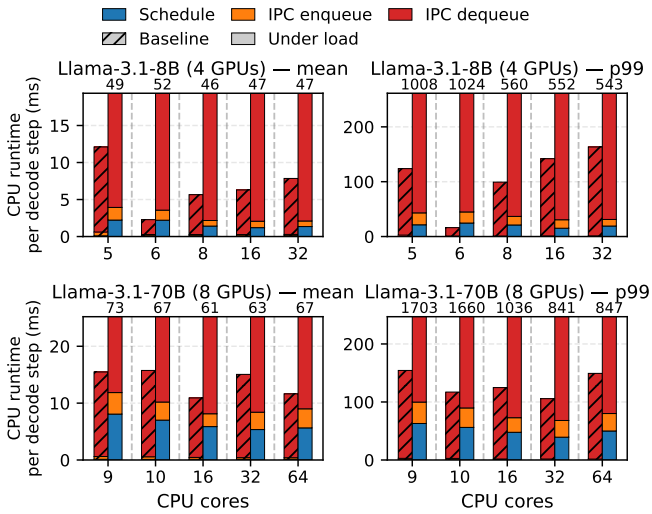

Fig. 12. CPU-side per-step runtime under baseline and load, decomposed into scheduling, IPC enqueue, and worker-side IPC dequeue. Rows show Llama 3.1 8B (4×H100, TP=4) and Llama 3.1 70B (8×H100, TP=8); columns show mean and p99 latency across CPU-core allocations.

number of GPU workers increases. Figure 11 illustrates how this contention can substantially delay model execution. In vLLM's V1 architecture, POSIX shared memory implements a broadcast queue that distributes scheduling decisions from EngineCore to each GPU worker process.

Results. Figure 12 decomposes the CPU-side work in a single vLLM V1 decode step under the same serving load used in Section IV. We measure the latency of three host-side components: the engine scheduler, the engine-to-worker shared-memory enqueue, and the worker-side shared-memory dequeue. The dequeue measurement includes time spent waiting for the next scheduling message. To avoid counting idle inter-arrival gaps as inter-process communication (IPC) overhead, we filter baseline dequeue waits above 200 ms. The plotted time is therefore CPU-side control-plane time only; it excludes the GPU model forward pass.

The mean CPU-side per-step runtime grows from roughly 2–16 ms at baseline to 46–73 ms under load, depending on model size and CPU allocation. This increase persists even at larger CPU allocations, showing that CPU-side scheduling and IPC overheads remain significant under sustained load. The tail is more severe: the component-wise p99 grows to 0.54–1.70 s under load, with worker-side `dequeue()` dominating the CPU-side critical path (91–96% of the p99 stack across all core allocations). This is a tail-latency effect rather than a shift in every iteration; typical steps remain much closer to baseline, but occasional delayed dequeues are enough to stall GPU workers and inflate request-level TTFT [12]. Unlike the mean, the tail responds to CPU provisioning: p99 falls by roughly 2× from (#GPUs + 1) cores to the most CPU-abundant allocation. However, additional CPU cores only mitigate these tail stalls; they do not remove the underlying synchronization dependency in the broadcast path.

Generality across serving frameworks. We inspected the corresponding control paths in SGLang v0.5.16 and TensorRT-

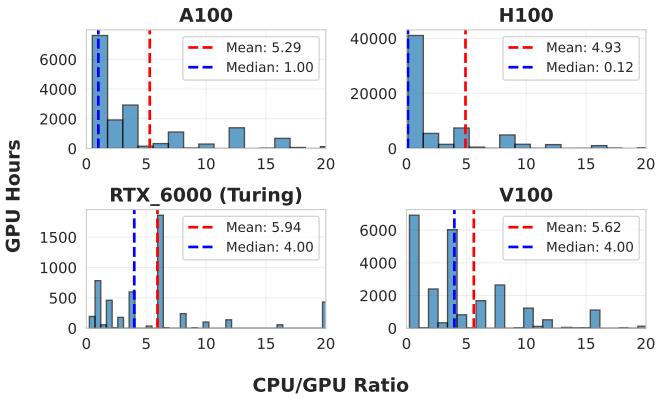

Fig. 13. GPU-hour-weighted CPU-to-GPU ratios on the instructional cluster (users manually set counts).

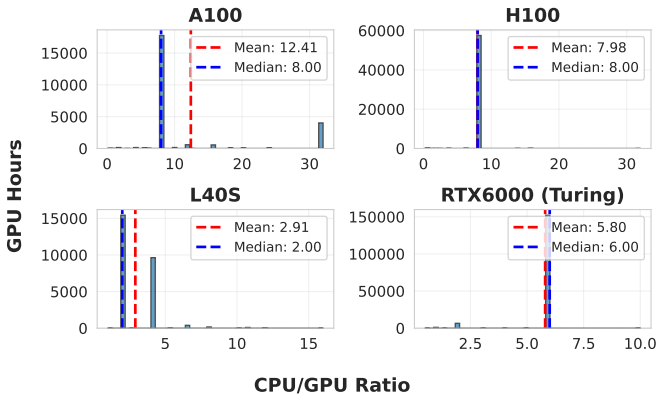

Fig. 14. GPU-hour-weighted CPU-to-GPU ratios on the research cluster (proportional unless overridden).

LLM v1.3.0, and both replicate the same one-writer, N-reader broadcast once per engine iteration: SGLang's rank-0 scheduler distributes batch metadata over a gloo CPU process group (`broadcast_pyobj()`) in a loop that neither sleeps nor yields, and TensorRT-LLM's rank 0 issues an MPI broadcast over the tensor-parallel communicator every iteration. Only the transport differs; in all three systems, a delayed host process cannot consume the broadcast, and the back-pressure stalls subsequent iterations. Therefore, we claim that the bottleneck is a structural consequence of driving N GPU workers from a replicated CPU control loop, independent of the engine or the actual implementation.

Takeaways. While additional CPU cores mitigate oversubscription-induced slowdowns, broadcast contention is a structural property of the 1-writer-N-reader pattern that adding cores alone cannot fully resolve. Even with adequate CPU cores, the writer must wait for the slowest reader before proceeding. This wait also grows with the number of GPUs allocated to the same LLM engine. The issue compounds in multi-tenant environments where tokenization and IPC polling compete for the same CPU cores.

Potential solutions. Mitigating these bottlenecks may require redesigned IPC mechanisms, such as asynchronous scheduling pipelines that overlap IPC with GPU execution. Examples include persistent GPU kernels that poll a device-side queue to eliminate per-step launch overhead [23], [39], and direct GPU-to-GPU signaling that bypasses the CPU control plane. Each of these comes at a cost: a persistent-kernel design, for example, requires restructuring the serving engine around a finer-grained CPU–GPU dependency chain, since the host must continuously feed a device-side queue instead of issuing discrete per-step launches. We therefore leave the design of practical mechanisms to future work.

VI. Discussion

A. Case Study: CPU Underprovisioning in HPC (High-Performance Computing) Clusters

As a concrete deployment case study, we analyze real-world resource allocation patterns from production HPC clusters. Prior cluster trace studies have characterized GPU job workloads at scale [25], [29], [78], but have not specifically examined CPU-to-GPU allocation ratios. We find that CPUs are indeed frequently under-requested, suggesting that imbalances between CPUs and GPUs are a practical performance problem.

We collected job scheduler logs from two institutional clusters operating within a university computing environment during calendar year 2024, totaling 4.65 million `salloc` records. The first cluster primarily serves research workloads, while the second supports instructional and educational use. Together, the clusters comprise approximately 1,400 compute nodes with around 35,000 CPU cores and several hundred GPUs, including recent high-end models such as NVIDIA H200 and RTX Pro 6000 Blackwell GPUs. The clusters serve heterogeneous CPU–GPU workloads, including ML training and inference as well as fluid-dynamics and computational-chemistry simulations, most of which run as multi-GPU jobs that also require multiple CPU workers per job.

Resource Allocation Policies. On the *research cluster*, the scheduler enforces a fixed CPU-to-GPU allocation policy: each GPU receives roughly $1/N$ of the node's total CPU cores, where N is the number of GPUs per node. This configuration preserves CPU/GPU affinity and prevents users from monopolizing CPU cores. In contrast, the *instructional cluster* does not enforce a strict CPU-to-GPU ratio, reflecting its instructional goal: students are allowed to tune the allocation strategies. On this cluster, the default Slurm scheduling parameter `--cpus-per-task=1` allocates only a single CPU core per task, severely degrading the performance of multi-GPU workloads.

Results. Figure 13 summarizes the GPU-hour-weighted distribution of CPU-to-GPU allocation ratios in the *instructional cluster*. We observe that many GPU-hours are concentrated at low CPU-to-GPU ratios, indicating that users often allocate far fewer CPU cores than GPUs. These low CPU allocations may partly reflect users forgetting or being unaware that additional CPU cores must be explicitly requested. Notably, H100 nodes, accounting for 34.3k GPU hours out of 50.9k total, include cases in which users request as few as 1 CPU core across 4 or 8 GPUs. These findings suggest that many users remain

unaware that inadequate CPU allocation can introduce severe bottlenecks in multi-GPU workloads.

Figure 14 shows the corresponding GPU-hour-weighted distribution for the *research cluster*. Low CPU-to-GPU ratios are less pronounced because the scheduler enforces proportional CPU allocation. Nevertheless, the distribution still includes many allocations with fewer than 8 CPU cores per GPU. As our experiments in Section IV show, ratios below 4–8 cores per GPU risk measurable slowdown for LLM inference workloads, leaving these jobs vulnerable to CPU-side performance bottlenecks.

Takeaways. While CPU scarcity is rare for large jobs that occupy an entire multi-GPU node (e.g., a full DGX box), it is common in shared-node workloads where users run multi-GPU jobs with limited CPU allocations. The problem becomes more pronounced when the server operator fails to force the system to allocate sufficient CPU resources per requested GPU. These observations corroborate the CPU-induced bottlenecks identified in Sections IV and V.

B. CPU Underprovisioning in Cloud Compute Platforms

Public cloud providers such as AWS offer GPU instances with pricing that highlights a large cost disparity between CPU and GPU resources [3]. GPU instances range from $3.06/hour for a single V100 (p3.2xlarge) to $55.04/hour for an 8×H100 configuration (p5.48xlarge) [3]. In contrast, CPU cores cost approximately $0.03–$0.06/hour per vCPU, making GPU compute roughly 100–200× more expensive than CPU cores depending on generation. Cloud instance configurations from major vendors commonly provide as few as 3–6 vCPUs per GPU [3], and the cluster-log case study above shows that users can allocate even fewer in shared environments.

Since the marginal cost of additional CPU cores is small relative to GPU instance pricing, e.g., adding 16 vCPUs at $0.05/hour each ($0.80/hour) to a p5.48xlarge instance at $55.04/hour represents roughly a 1.5% cost increase, our analysis shows that adding CPU resources is remarkably cost-effective. For CPU-bound workloads, performance scales nearly linearly with additional cores at minimal cost, reducing the total GPU hours required and delivering better overall throughput per dollar than simply provisioning more GPUs. This cost argument extends to ARM-based hosts such as NVIDIA Grace. Grace Hopper systems remain host-bound across larger batch-size regimes than `x86` hosts [18], [73], so the CPU-side bottlenecks we characterize are likely to persist on these platforms as well. Meanwhile, ARM host cores carry a lower per-core cost than `x86` server cores, making CPU overprovisioning even more cost-effective relative to GPU pricing; vendors already ship generous ratios by default, such as 72 host cores per GPU on GH200 [51].

We note that conventional host-side mitigation strategies do not fully address the CPU bottlenecks identified in this work. Admission control and rate limiting can prevent CPU overload. However, they do so by reducing the accepted query rate, leaving GPU capacity underutilized and undermining the economics of multi-GPU deployment. OS-level resource isolation via `cgroup`, `cpuset`, and CPU pinning tools such as `taskset` and `isolcpus` [16] can improve scheduling determinism by dedicating cores to latency-sensitive processes, but cannot compensate when the total number of allocated cores is fundamentally insufficient for the workload's concurrency demands. Accordingly, our evaluation focuses on aggregate CPU capacity rather than process-affinity policies. While large cloud instances (e.g., AWS p5.48xlarge) provision up to 24 vCPUs per GPU by default, multi-tenant sharing and cost-driven optimization frequently reduce effective per-GPU CPU allocations well below this level.

More fundamentally, solutions to CPU bottlenecks fall into two categories: *host-driven* approaches that strengthen the CPU (adding cores, pinning, priority scheduling) and *device-initiated* approaches that remove the CPU from the critical path entirely, such as GPU-initiated networking [24], [28], [55], [67], storage access [41], [54], [60], and persistent GPU kernels [39]. Our results quantify the benefits of adding CPU cores; the latter represents a promising but more invasive architectural direction. Notably, modern GPUs already include on-die control processors, such as NVIDIA's RISC-V-based GPU System Processor (GSP) [1], which potentially can take over the orchestration duties currently handled by the host.

C. Emerging Trends That Intensify the CPU Bottleneck

Several emerging trends in LLM deployment suggest that CPU-side bottlenecks will become more pronounced rather than less. First, *agentic AI* workloads perform dynamic reasoning with frequent tool invocations, each requiring CPU-side processing that creates substantial GPU idle time [2], [32], [61], [93]. Moreover, *long-context inference*, combined with *frequent prefills* produced by multi-turn agentic tool invocations, imposes substantial load on the tokenizer, potentially consuming several seconds of CPU time for each request. As most newly introduced LLMs now natively support context lengths over 1M tokens, the tokenizer's CPU bottleneck will become even more significant [4], [33], [70].

Multimodal inputs such as images and video require CPU-intensive preprocessing (resizing, normalization, frame extraction) before GPU inference can begin, adding further pressure to the host. Finally, as GPU compute scales faster than CPU performance, deploying even moderately sized models across multiple GPUs reduces per-GPU compute time, increasing the relative contribution of CPU coordination overhead. Together, these trends indicate that the CPU bottlenecks characterized in this work are not artifacts of current-generation systems but structural challenges that will require architectural attention.

D. Limitations

Our study is bounded by the hardware and frameworks we could access. All experiments use NVIDIA GPUs with Intel CPUs and SMT disabled; production cloud instances typically expose vCPUs, so our physical-core measurements form a conservative baseline. Our findings may differ on CPU-GPU coupled architectures such as Grace Hopper [51], where ARM

cores show different kernel launch characteristics [73], or high-core-count AMD EPYC hosts. Our quantitative evaluation centers on vLLM; while our structural analysis covers the control paths of SGLang [92] and TensorRT-LLM [53], full quantitative validation across frameworks remains future work.

VII. Related Work

A. CPU Overhead in Agentic AI

Agentic AI workloads with dynamic reasoning and tool usage increasingly become CPU-bound [42], [69]. Kim et al. [32] show that frequent tool invocations cause substantial GPU idle time, shifting workloads to CPU- or I/O-bound tasks. Raj et al. [61] find that tool-processing components run predominantly on CPUs, accounting for up to 90.6% of total latency.

B. Host-Side LLM Inference Characterization

Recent profiling studies quantify host-side overheads and scheduling challenges in agentic LLM inference [33], [83]. Vellaisamy et al. [73] introduce SKIP profiler and the total kernel launch and queuing time (TKLQT) metric, showing that CPU-GPU coupled systems such as GH200 remain CPU-bound across larger batch-size regions where single-thread CPU performance and kernel launch overhead constrain low-batch latency. TaxBreak [74] decomposes host-visible orchestration overhead and shows that host-bound workloads, especially MoE inference, are sensitive to CPU single-thread performance; faster host CPUs reduce orchestration overhead by 10–29% and improve end-to-end latency by up to 14%. These studies diagnose single-node host overheads; our work is complementary, focusing on concurrent multi-GPU serving where tokenization, IPC broadcast, and barrier-based synchronization compound into TTFT slowdowns and timeouts.

C. GPU-Driven Control and Device-Initiated Communications

GPU-driven execution and device-initiated communication aim to remove the CPU from the critical path. ARK [28] reduces CPU involvement by enabling GPU threads to directly control DMA operations. NVSHMEM [55] enables device-initiated communication, with NCCLX [67] parallelizing remote direct memory access (RDMA) setup via lightweight GPU threads and DeepEP [39] managing MoE expert parallelism [30]. Yet widely used frameworks such as vLLM and PyTorch still rely on CPU-driven coordination. LithOS [8] redesigns the OS-GPU interface by atomizing long-running kernels for preemptive scheduling, reducing control-plane head-of-line blocking. Other work leverages spare CPU resources for distributed training [31].

D. LLM Serving Frameworks

Modern LLM serving systems adopt multi-process architectures to isolate CPU-side tasks from GPU execution, including vLLM [36] (Section III), SGLang [92], and TensorRT-LLM [53]. DistServe [94] and Splitwise [56] disaggregate the prefill and decode phases onto separate GPU pools to optimize goodput, thereby concentrating CPU-intensive tokenization on dedicated prefill servers. While these frameworks optimize GPU utilization and scheduling, all rely on CPU-side processes for tokenization, request management, and inter-process coordination, leaving them vulnerable to the bottlenecks characterized in this work.

E. Mitigating the CPU Overhead in LLMs

Tokenization is a key CPU bottleneck in LLMs. NetTokenizer [44] offloads tokenization to programmable network data planes, BlockBPE [85] runs BPE on GPUs (effective for high-batch but not latency-sensitive workloads), and LoPT [65] achieves lossless parallel tokenization via token merging. Recasens et al. [63] report CPU overhead reaching up to 30% of total execution time in large-batch LLM inference.

F. Mitigating Kernel Launch Overhead

Lustig et al. [43] propose early kernel launch to reduce CPU-GPU synchronization latency, and CPU pinning reduces scheduling jitter [16]. CUDA Graphs [15], [47] reduce launch overhead for static sequences but cannot capture dynamic LLM decoding decisions; Grape [91] and KTransformers [6] extend them to dynamic DNNs and MoE models, respectively, but neither addresses tokenization contention. FlashAttention [10] fuses kernels but does not address launch-path overhead.

Prior works have independently noted CPU overhead in tokenization [65], [85], host orchestration and kernel launch paths [28], [73], [74], and data preprocessing pipelines [45], while complementary GPU-side characterizations [63], [75] focus on compute and memory bottlenecks within the accelerator. None of these works examines how these factors compound under multi-GPU LLM serving, where tokenization contention, IPC broadcast delays, and barrier-based synchronization interact to produce cascading slowdowns. Our work systematically characterizes the CPU-side effects that compound with these GPU-side phenomena and quantifies how CPU provisioning affects end-to-end performance.

VIII. Conclusion

Our study shows that CPU-side responsibilities can become critical bottlenecks in multi-GPU LLM inference, even in fully optimized serving stacks. CPU oversubscription, combined with barrier-based GPU synchronization, leaves GPUs idle, while shared-memory broadcast contention introduces structural delays proportional to the degree of tensor parallelism. Providing adequate CPU resources improves TTFT by 1.47–7.11$\times$ at little marginal cost relative to GPUs, and our analysis of 4.65 million cluster job records confirms widespread under-provisioning.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback. This work was supported in part by an NSF grant CCF-2316176. Claude Opus 4.7 was used for grammar checking and assistance with experimental environment setup; the authors take full responsibility for all content and results.

REFERENCES

[1] "NVIDIA GSP firmware," https://download.nvidia.com/XFree86/Linux-x86_64/560.28.03/README/gsp.html, 2024.

[2] R. Abhyankar, Z. He, V. Srivatsa, H. Zhang, and Y. Zhang, "Infercept: Efficient intercept support for augmented large language model inference," *arXiv preprint arXiv:2402.01869*, 2024.

[3] "Amazon ec2 instance types," https://aws.amazon.com/ec2/instance-types/, AWS.

[4] A. Blakeman, A. Grattafiori, A. Basant, A. Gupta, A. Khattar, A. Renduchintala, A. Vavre, A. Shukla, A. Bercovich, A. Ficek *et al.*, "Nvidia nemotron 3: Efficient and open intelligence," *arXiv preprint arXiv:2512.20856*, 2025.

[5] C. Chang, Y. Zhou, K. Fu, D. An, T. Feng, H. Lu, S. Yao, P. Guo, Y. Yu, Y. Shan *et al.*, "From llm inference to agentic workloads: Characterization and implications for serving systems," *arXiv preprint arXiv:2608.15127*, 2026.

[6] H. Chen, W. Xie, B. Zhang, J. Tang, J. Wang, J. Dong, S. Chen, Z. Yuan, C. Lin, C. Qiu *et al.*, "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*, 2025, pp. 1014–1029.

[7] E. Chung, S. Na, S. H. Kang, and H. Kim, "Swift and trustworthy large-scale gpu simulation with fine-grained error modeling and hierarchical clustering," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, 2025, pp. 1397–1411.

[8] P. H. Coppock, B. Zhang, E. H. Solomon, V. Kypriotis, L. Yang, B. Sharma, D. Schatzberg, T. C. Mowry, and D. Skarlatos, "Lithos: An operating system for efficient machine learning on gpus," in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, 2025, pp. 1–17.

[9] Crusoe Cloud, "fastokens," https://github.com/crusoecloud/fastokens, 2026.

[10] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in neural information processing systems*, vol. 35, pp. 16 344–16 359, 2022.

[11] M. Davies, N. Crago, K. Sankaralingam, and C. Kozyrakis, "Liminal: Exploring the frontiers of llm decode performance," *arXiv preprint arXiv:2507.14397*, 2025.

[12] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[13] J. Ding, R. Hosseini, P. M. Gholami, M. Xiang, and H. Hoffmann, "Observation, not prediction: Conversation-level disaggregated scheduling for agentic serving," *arXiv preprint arXiv:2606.01839*, 2026.

[14] S. Durvasula, A. Zhao, R. Kiguru, Y. Guan, Z. Chen, and N. Vijaykumar, "Acs: Concurrent kernel execution on irregular, input-dependent computational graphs," *arXiv preprint arXiv:2401.12377*, 2024.

[15] J. Ekelund, S. Markidis, and I. Peng, "Boosting performance of iterative applications on gpus: Kernel batching with cuda graphs," in *2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE, 2025, pp. 70–77.

[16] "Os-level challenges in LLM inference and optimizations," https://eunomia.dev/blog/2025/02/18/os-level-challenges-in-llm-inference-and-optimizations/, Eunomia, 2025.

[17] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022.

[18] L. Fusco, M. Khalilov, M. Chrapek, G. Chukkapalli, T. Schulthess, and T. Hoefler, "Understanding data movement in tightly coupled heterogeneous systems: A case study with the grace hopper superchip," *arXiv preprint arXiv:2408.11556*, 2024.

[19] B. Gao, Z. He, P. Sharma, Q. Kang, D. Jevdjic, J. Deng, X. Yang, Z. Yu, and P. Zuo, "{Cost-Efficient} large language model serving for multi-turn conversations with {CachedAttention}," in *2024 USENIX annual technical conference (USENIX ATC 24)*, 2024, pp. 111–126.

[20] I. Gim, G. Chen, S.-s. Lee, N. Sarda, A. Khandelwal, and L. Zhong, "Prompt cache: Modular attention reuse for low-latency inference," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 325–338, 2024.

[21] S. Gugger, L. Debut, T. Wolf, P. Schmid, Z. Mueller, S. Mangrulkar, M. Sun, and B. Bossan, "Accelerate: Training and inference at scale made simple, efficient and adaptable." https://github.com/huggingface/accelerate, 2022.

[22] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving {DNNs} like clockwork: Performance predictability from the bottom up," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 443–462.

[23] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *2012 Innovative Parallel Computing (InPar)*. IEEE, 2012, pp. 1–14.

[24] K. Hamidouche, J. Bachan, P. Markthub, P.-J. Gootzen, E. Agostini, S. Jeaugey, A. Shafi, G. Theodorakis, and M. G. Venkata, "Gpu-initiated networking for nccl," *arXiv preprint arXiv:2511.15076*, 2025.

[25] Q. Hu, Z. Ye, Z. Wang, G. Wang, M. Zhang, Q. Chen, P. Sun, D. Lin, X. Wang, Y. Luo *et al.*, "Characterization of large language model development in the datacenter," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 709–729.

[26] Z. Hu, S. Shen, T. Bonato, S. Jeaugey, C. Alexander, E. Spada, J. Dinan, J. Hammond, and T. Hoefler, "Demystifying nccl: An in-depth analysis of gpu communication protocols and algorithms," *arXiv preprint arXiv:2507.04786*, 2025.

[27] "Huggingface tokenizers," https://huggingface.co/docs/tokenizers/index, HuggingFace.

[28] C. Hwang, K. Park, R. Shu, X. Qu, P. Cheng, and Y. Xiong, "{ARK}:{GPU-driven} code execution for distributed deep learning," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 87–101.

[29] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 947–960.

[30] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv preprint arXiv:2401.04088*, 2024.

[31] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 463–479.

[32] J. Kim, B. Shin, J. Chung, and M. Rhu, "The cost of dynamic reasoning: Demystifying ai agents and test-time scaling from an ai infrastructure perspective," *arXiv preprint arXiv:2506.04301*, 2025.

[33] L. Kondrashov, H. Liu, J. Park, B. Zhou, Z. Liu, C. Lu, R. Mancini, E. Choukse, H. Javaid, G. Sviridov *et al.*, "Rethinking ai cloud infrastructure for agentic serving systems with the aries experimentation framework," *arXiv preprint arXiv:2607.29069*, 2026.

[34] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[35] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," *arXiv preprint arXiv:1808.06226*, 2018.

[36] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. 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*, 2023, pp. 611–626.

[37] H. Li, R. He, Q. Mang, Q. Zhang, H. Mao, X. Chen, H. Zhou, A. Cheung, J. Gonzalez, and I. Stoica, "Continuum: Efficient and robust multi-turn llm agent scheduling with kv cache time-to-live," *arXiv preprint arXiv:2511.02230*, 2025.

[38] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[39] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[40] B. Liu, H. Qiu, Í. Goiri, R. Fonseca, R. Bianchini, and E. Choukse, "Agentic coding in the wild: Characterizing github copilot traces at production scale," *arXiv preprint arXiv:2608.00101*, 2026.

[41] X. Liu, S. Na, E. Chung, J. Cao, J. Yang, and H. Kim, "Contention-aware gpu thread block scheduler for efficient gpu-ssd," *IEEE Computer Architecture Letters*, 2025.

[42] M. Luo, X. Shi, C. Cai, T. Zhang, J. Wong, Y. Wang, C. Wang, Y. Huang, Z. Chen, J. E. Gonzalez *et al.*, "Autellix: An efficient serving engine for llm agents as general programs," *arXiv preprint arXiv:2502.13965*, 2025.

[43] D. Lustig and M. Martonosi, "Reducing gpu offload latency via fine-grained cpu-gpu synchronization," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013, pp. 354–365.

[44] S. Mehta, "Accelerating llm inference with smart nic tokenization and caching," 2025.

[45] J. Mohan, A. Phanishayee, J. Kulkarni, and V. Chidambaram, "Looking beyond {GPUs} for {DNN} scheduling on {Multi-Tenant} clusters," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 579–596.

[46] C. Nie, N. Si, and Z. Zhou, "A queueing-theoretic framework for stability analysis of llm inference with kv cache memory constraints," *arXiv preprint arXiv:2605.04595*, 2026.

[47] "Getting started with cuda graphs," https://developer.nvidia.com/blog/cuda-graphs/, NVIDIA.

[48] "Nvidia collective communications library (nccl)," https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html, NVIDIA.

[49] "Nvidia dgx b200," https://www.nvidia.com/en-us/data-center/dgx-b200/, NVIDIA.

[50] "Nvidia dgx h100," https://www.nvidia.com/en-gb/data-center/dgx-h100/, NVIDIA.

[51] "Nvidia gh200 grace hopper superchip," https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/, NVIDIA.

[52] "Nvidia dali: A gpu-accelerated data loading and pre-processing library," https://github.com/NVIDIA/DALI, NVIDIA, 2017.

[53] "Tensorrt-llm," https://github.com/NVIDIA/TensorRT-LLM, NVIDIA, 2023.

[54] NVIDIA, "Nvidia gpudirect storage," https://docs.nvidia.com/gpudirect-storage/, 2024.

[55] "Nvshmem," https://developer.nvidia.com/nvshmem, NVIDIA, 2024.

[56] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 118–132.

[57] "Scaling analysis," https://researchcomputing.princeton.edu/support/knowledge-base/scaling-analysis, Princeton Research Computing.

[58] "torch.profiler – pytorch 2.9 documentation," https://docs.pytorch.org/docs/stable/profiler.html, Pytorch.

[59] R. Qin, Z. Li, W. He, J. Cui, H. Tang, F. Ren, T. Ma, S. Cai, Y. Zhang, M. Zhang *et al.*, "Mooncake: A kvcache-centric disaggregated architecture for llm serving," *ACM Transactions on Storage*, 2024.

[60] Z. Qureshi, V. S. Mailthody, I. Gelado, S. Min, A. Masood, J. Park, J. Xiong, C. J. Newburn, D. Vainbrand, I.-H. Chung *et al.*, "Gpu-initiated on-demand high-throughput storage access in the bam system architecture," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 325–339.

[61] R. Raj, H. Wang, and T. Krishna, "A cpu-centric perspective on agentic ai," *arXiv preprint arXiv:2511.00739*, 2025.

[62] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "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*, 2020, pp. 3505–3506.

[63] P. G. Recasens, F. Agullo, Y. Zhu, C. Wang, E. K. Lee, O. Tardieu, J. Torres, and J. L. Berral, "Mind the memory gap: Unveiling gpu bottlenecks in large-batch llm inference," *arXiv preprint arXiv:2503.08311*, 2025.

[64] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: long papers)*, 2016, pp. 1715–1725.

[65] W. Shao, L. Zheng, P. Wang, P. Zheng, J. Li, and Y. Fan, "Lopt: Lossless parallel tokenization acceleration for long context inference of large language model," 2025. [Online]. Available: https://arxiv.org/abs/2511.04952

[66] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[67] M. Si, P. Balaji, Y. Chen, C.-H. Chu, A. Gangidi, S. Hasan, S. Iyengar, D. Johnson, B. Liu, J. Ren *et al.*, "Collective communication for 100k+ gpus," *arXiv preprint arXiv:2510.20171*, 2025.

[68] "Slurm documentation," https://slurm.schedmd.com/documentation.html, Slurm.

[69] V. Srivatsa, Z. He, R. Abhyankar, D. Li, and Y. Zhang, "Preble: Efficient distributed prompt scheduling for llm serving," in *International conference on learning representations*, vol. 2025, 2025, pp. 37 057–37 082.

[70] Q. Team, "Qwen3. 5-omni technical report," *arXiv preprint arXiv:2604.15804*, 2026.

[71] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, "Training data-efficient image transformers & distillation through attention," 2021.

[72] R. Varra, S. Ashtikar, V. Lal, and S. Krishnapura, "The rising cpu:gpu ratio in ai infrastructure: Drivers, trends, and implications," Intel, 2026.

[73] P. Vellaisamy, T. Labonte, S. Chakraborty, M. Turner, S. Sury, and J. P. Shen, "Characterizing and optimizing llm inference workloads on cpu-gpu coupled architectures," in *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2025, pp. 49–61.

[74] P. Vellaisamy, S. Tripathi, V. Natarajan, S. S. Thenarasu, S. Blanton, and J. P. Shen, "Taxbreak: Unmasking the hidden costs of llm inference through overhead decomposition," in *2026 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2026.

[75] H. Wang, X. Xiao, M. Yan, Z. Zhu, D. Han, D. Wang, W. Li, X. Ye, C. Hu, H. Chen *et al.*, "A systematic characterization of llm inference on gpus," *arXiv preprint arXiv:2512.01644*, 2025.

[76] X. Wang, M. Salmani, P. Omidi, X. Ren, M. Rezagholizadeh, and A. Eshaghi, "Beyond the limits: A survey of techniques to extend the context length in large language models," *arXiv preprint arXiv:2402.02244*, 2024.

[77] Y. Wang, Y. Chen, Z. Li, X. Kang, Z. Tang, X. He, R. Guo, X. Wang, Q. Wang, A. C. Zhou *et al.*, "Burstgpt: A real-world workload dataset to optimize llm serving systems.(2024)," *URL https://arxiv. org/abs/2401*, vol. 17644, 2024.

[78] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "{MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 945–960.

[79] M. Wong, U. Butler, E. Farkash, P. Tammana, A. Sivaraman, and R. Netravali, "Gpus, cpus, and... nics: Rethinking the network's role in serving complex ai pipelines," *arXiv preprint arXiv:2502.15712*, 2025.

[80] B. Wu, C. Xu, X. Dai, A. Wan, P. Zhang, Z. Yan, M. Tomizuka, J. Gonzalez, K. Keutzer, and P. Vajda, "Visual transformers: Token-based image representation and processing for computer vision," 2020.

[81] Y. Wu, S. Chen, Y. Zhong, R. Huang, Y. Tan, W. Zhang, L. Zhang, S. Zhou, Y. Liu, S. Zhou *et al.*, "Dualpath: breaking the storage bandwidth bottleneck in agentic llm inference," *arXiv preprint arXiv:2602.21548*, 2026.

[82] Y. Xiang, X. Li, K. Qian, Y. Zhang, W. Yu, E. Zhai, X. Jin, and J. Zhou, "{ServeGen}: Workload characterization and generation of large language model serving in production," in *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, 2026, pp. 1845–1859.

[83] J. Yang, P. Liu, C. Zhang, and J. Stojkovic, "Architectural implications of agentic ai workflows," *arXiv preprint arXiv:2608.04458*, 2026.

[84] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.

[85] A. You, "Blockbpe: Parallel bpe tokenization," *arXiv preprint arXiv:2507.11941*, 2025.

[86] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for {Transformer-Based} generative models," in *16th USENIX symposium on operating systems design and implementation (OSDI 22)*, 2022, pp. 521–538.

[87] L. Yu, J. Lin, and J. Li, "Stateful large language model serving with pensieve," in *Proceedings of the Twentieth European Conference on Computer Systems*, 2025, pp. 144–158.

[88] Y. Yuan, M. Chowdhury, and N. Talati, "Kairos: Stateful, context-aware power-efficient agentic inference serving," *arXiv preprint arXiv:2604.16682*, 2026.

[89] Z. Zhang and Z. Cao, "Toktier: Exact stateful tokenization for agentic llm serving," *arXiv preprint arXiv:2607.29678*, 2026.

[90] C. Zhao, C. Deng, C. Ruan, D. Dai, H. Gao, J. Li, L. Zhang, P. Huang, S. Zhou, S. Ma *et al.*, "Insights into deepseek-v3: Scaling challenges and reflections on hardware for ai architectures," in *Proceedings of the*

52nd Annual International Symposium on Computer Architecture, 2025, pp. 1731–1745.

[91] B. Zheng, C. H. Yu, J. Wang, Y. Ding, Y. Liu, Y. Wang, and G. Pekhimenko, "Grape: Practical and efficient graphed execution for dynamic deep neural networks on gpus," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1364–1380.

[92] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez *et al.*, "Sglang: Efficient execution of structured language model programs," *Advances in neural information processing systems*, vol. 37, pp. 62 557–62 583, 2024.

[93] Y. Zheng, J. Fan, Q. Fu, Y. Yang, W. Zhang, and A. Quinn, "Agentcgroup: Understanding and controlling os resources of ai agents," *arXiv preprint arXiv:2602.09345*, 2026.

[94] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "{DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 193–210.

[95] K. Zhu, M. Jacob, C. Ma, Y. Pan, S. Wang, A. Krishnamurthy, and B. Kasikci, "Tracelab: Characterizing coding agent workloads for llm serving," *arXiv preprint arXiv:2606.30560*, 2026.