Towards Multi-Model LLM Schedulers: Empirical Insights into Offloading and Preemption

Mert Yildiz*, Pietro Spadaccino*, Alexey Rolich*, Francesca Cuomo*, Andrea Baiocchi*
*Sapienza University of Rome, Rome, Italy
Email: {mert.yildiz, pietro.spadaccino, alexey.rolich, francesca.cuomo, andrea.baiocchi}@uniroma1.it

2026 Mediterranean Artificial Intelligence and Networking Conference (MAIN) | 979-8-3195-2594-9/26/$31.00 ©2026 IEEE | DOI: 10.1109/MAIN71116.2026.11622343

***Abstract*—Modern deployments of Large Language Models (LLMs) increasingly require serving multiple models with diverse architectures, sizes, and specialization on shared, heterogeneous hardware. This setting introduces new challenges for resource allocation, dispatching, and scheduling, particularly under GPU memory constraints where partial CPU–GPU offloading and preemption become necessary. While existing systems primarily optimize throughput for a single model, comparatively little work addresses multi-model scheduling under these conditions. In this paper, we present an empirical study of how different LLMs behave across hardware platforms, focusing on the performance implications of layer offloading and preemption. We show that offloading leads to strongly non-linear and model-dependent degradation in decode throughput, with smaller models exhibiting sharper sensitivity to reduced GPU residency. We further demonstrate that preemption incurs substantial overhead, largely dominated by model state reload rather than key-value cache transfer, and that this cost varies significantly across models and hardware platforms. Additionally, we highlight the role of sequence length and interconnect bandwidth in amplifying data movement and execution inefficiencies. Based on these findings, we identify a set of key features that future schedulers must consider, including model-specific offloading sensitivity, workload characteristics, and the cost structure of preemption and data transfer. These insights provide guidance for the design of next-generation LLM serving systems capable of efficiently managing heterogeneous, multi-model workloads with hybrid CPU–GPU execution.**

***Index Terms*—LLM inference; CPU-GPU offloading; KV cache migration; Inference scheduling; Multi model; Multimodal**

I. Introduction

In recent years, large language models (LLMs) have become a central component of modern artificial intelligence systems, enabling advanced capabilities in natural language understanding, generation, and reasoning. The rapid pace of model development has led to a diverse ecosystem of LLMs that differ in architecture, size, training data, and performance characteristics. As a result, real-world deployments increasingly involve not a single model, but a collection of heterogeneous models that are continuously updated, replaced, or specialized for different tasks.

In this field, practitioners routinely evaluate and deploy multiple models to balance trade-offs between accuracy, latency, and resource consumption. In both research and industrial settings, it is common to select different models depending on application requirements, system constraints, or cost considerations. Moreover, the ecosystem of LLMs is increasingly specialized: some models are optimized for natural language generation, others for code synthesis, and others for agentic or tool-augmented reasoning. This diversity motivates the need for systems capable of serving multiple models concurrently, as no single model uniformly dominates across all tasks and deployment scenarios.

Managing such heterogeneous workloads requires efficient mechanisms for resource allocation, request dispatching, and scheduling across available hardware. These challenges are exacerbated in environments with limited GPU resources, where multiple models compete for memory and compute, and where dynamic workload patterns introduce variability in demand. Effective system design must therefore account for both inter-model heterogeneity and intra-model execution characteristics, while ensuring high utilization and predictable latency.

Existing LLM serving systems have primarily focused on optimizing throughput for a single model by batching and executing multiple requests in parallel. Systems such as vLLM [1] and related frameworks exploit techniques like continuous batching and efficient key-value cache management to maximize GPU utilization when serving many concurrent requests to the same model. While these approaches are effective in homogeneous settings, they do not directly address scenarios in which requests are distributed across multiple models with different resource footprints and execution profiles.

In contrast, comparatively little attention has been devoted to the problem of serving multiple LLMs concurrently within a shared infrastructure. In such settings, scheduling decisions must consider not only the number of requests, but also model-specific characteristics, partial offloading strategies, and the interaction between CPU and GPU resources. This introduces a fundamentally different set of trade-offs, where isolating models may simplify execution but reduce utilization, while sharing resources across models increases efficiency at the cost of greater scheduling complexity.

In this paper, we investigate the behavior of diverse LLMs, spanning different architectures and model scales, across heterogeneous hardware platforms. Our goal is to characterize how these models interact with system resources under varying execution configurations, including partial CPU-GPU offloading. Through this analysis, we identify the key factors that influence performance, memory usage, and latency across models and hardware settings. Building on these insights, we outline the requirements for next-generation scheduling systems, highlighting the considerations necessary to support advanced mechanisms such as preemption and hybrid CPU-GPU offloading. Our results provide guidance for the design of future schedulers capable of efficiently managing multi-model workloads in dynamic and resource-constrained environments.

A. LLM Computing Model

Large Language Models based on transformer architectures execute inference as a sequence of layered computations, where each layer performs a set of operations including linear projections, attention mechanisms, and non-linear transformations. These layers are strictly ordered and must be evaluated sequentially due to data dependencies between them. In

Authorized licensed use limited to: Universita degli Studi di Roma La Sapienza. Downloaded on October 27,2026 at 15:13:24 UTC from IEEE Xplore. Restrictions apply.

heterogeneous environments, individual layers can be mapped either to GPUs or CPUs. GPUs provide significantly higher throughput due to massive parallelism and specialized tensor cores, whereas CPUs offer lower computational performance but greater flexibility and memory capacity. Consequently, partial offloading strategies assign subsets of layers to the CPU to alleviate GPU memory pressure, at the cost of increased latency due to slower execution and potential data transfer overheads.

Inference in LLMs follows an autoregressive generation process, in which tokens are produced sequentially. At each decoding step, the model consumes the previously generated tokens and computes the probability distribution for the next token. This introduces an inherent temporal dependency across steps, preventing parallelization across tokens during generation. As a result, end-to-end latency is determined not only by the per-layer execution time but also by the cumulative effect of sequential token generation, making scheduling decisions sensitive to both computation placement and token-level iteration dynamics.

Memory consumption in LLM inference can be decomposed into a static and a dynamic component. The static component corresponds to model parameters, which remain fixed during inference and must reside in memory across all decoding steps. The dynamic component arises from intermediate activations and, critically, the key-value (KV) cache used in attention mechanisms. The KV cache stores representations of previously processed tokens to avoid redundant computation in subsequent steps, leading to memory growth proportional to the sequence length. This dynamic memory footprint significantly impacts resource allocation and scheduling, particularly when multiple concurrent inference requests are served or when operating under constrained GPU memory, thereby motivating hybrid CPU-GPU execution strategies.

Our contributions are as follows:

1) we conduct a systematic empirical evaluation of state-of-the-art LLMs across heterogeneous hardware platforms, demonstrating that key performance trends, including the impact of offloading and scaling behavior, remain consistent across models and systems;
2) we analyze the performance implications of partial CPU–GPU offloading and preemption, quantifying their effects on decode throughput, latency, and data movement, and identifying the dominant sources of overhead;
3) we distill these observations into a set of key features that future schedulers must consider, providing guidance for the design of scheduling policies that support multi-model workloads with hybrid CPU–GPU execution and preemption.

The remainder of this paper is organized as follows. Section II reviews related work. Section III describes the experimental setup. Sections IV and V present the offloading and preemption results respectively. Section VI distills the findings into key features for future schedulers, and conclusions are drawn in Section VII.

II. Related Work

Several systems address the case where model weights do not fit in GPU memory. FlexGen [2] formulates tensor placement across GPU, CPU, and disk as a linear program that maximizes batch throughput. LIA [3] leverages Intel AMX matrix-multiplication units on the CPU to cooperatively execute sublayers alongside the GPU, combined with CXL memory offloading, achieving up to 19× lower latency than prior offloading frameworks on OPT-175B. PowerInfer [4] exploits activation sparsity to keep only frequently activated neurons on the GPU. Fiddler [5] and KTransformers [6] extend heterogeneous execution to Mixture-of-Experts architectures. NEO [7] offloads attention and KV-cache operations to the CPU to relieve memory pressure during online serving, and LM-Offload [8] builds an analytical model to guide placement decisions. Each of these works evaluates its design at a handful of fixed configurations, typically comparing against a fully-GPU and a fully-CPU baseline. None systematically varies the fraction of offloaded layers as an independent variable, leaving open the question of how decode throughput degrades as a continuous function of the offloading ratio across different model sizes and hardware platforms.

Managing the KV cache, whose memory footprint grows linearly with sequence length [9], and deciding when to preempt or reschedule requests are tightly coupled problems that have driven much of the recent progress in LLM serving. Orca [10] introduces continuous batching, allowing requests to enter and leave a running batch at every decoding step. vLLM [1] manages KV caches via paged memory inspired by operating-system virtual memory, and when memory runs out, preempts requests by discarding their cache and later recomputing it. Sarathi [11], [12] piggybacks decode tokens onto prefill batches through chunked prefill, improving GPU utilization by mixing the two phases within a single iteration. Splitwise [13] and DistServe [14] disaggregate prefill and decode onto separate GPU pools to match each phase to its bottleneck resource, which requires migrating the KV cache over the network after prompt processing completes. DéjàVu [15] streams KV-cache state between GPU and CPU or across nodes to enable fast recovery from failures and low-overhead migration in distributed serving. InfiniGen [16] offloads KV-cache entries to CPU memory and speculatively prefetches only the entries needed for the next decoding step back to the GPU. InstInfer [17] pushes this further by offloading attention computation itself to CXL-attached storage devices, keeping the GPU free for other operations. Llumnix [18] introduces live migration of KV caches across GPU instances for runtime rescheduling and load balancing, analogous to virtual-machine live migration. FastSwitch [19] identifies that preemption-induced context switching can stall inference for longer than the useful computation itself, and proposes coarser-grained batched transfers to reduce this overhead.

FastServe [20] swaps KV-cache state to host memory over PCIe and schedules requests with a skip-join Multi-Level Feedback Queue that supports token-level preemption. S^3 [21] and Zheng et al. [22] predict output lengths to approximate shortest-job-first ordering without preemption. Kim et al. [23] adapt DBMS-inspired cost models to LLM inference and find that strategically preempting short requests under memory pressure can up to double throughput compared to preemption-avoidance policies. SpotServe [24] addresses preemption at a coarser grain, migrating entire inference jobs across preemptible cloud instances.

All of the systems discussed above assume that a single model occupies the GPU throughout the serving lifetime. In practice, inference clusters must serve requests targeting many different models, and multiplexing a GPU across models

Authorized licensed use limited to: Universita degli Studi di Roma La Sapienza. Downloaded on October 27,2026 at 15:13:24 UTC from IEEE Xplore. Restrictions apply.

remains an open challenge that recent surveys identify as central to scalable LLM deployment [25]. The dominant production approach, exemplified by vLLM [1], binds one model per engine instance and handles multi-model deployments by running separate instances behind a request router, leaving model weights static in GPU memory and managing memory pressure solely through KV cache eviction or recomputation. MuxServe [26] relaxes this constraint by colocating multiple LLMs on the same GPU through spatial partitioning of streaming multiprocessors via NVIDIA MPS, combined with temporal interleaving of prefill and decode phases. However, co-residency is hard-limited by GPU memory capacity: with typical model footprints of 20–30 GB in FP16, at most two to three models can share an 80 GB GPU simultaneously [27]. Prism [28] introduces a two-level scheduler that places models across GPUs based on resource demands and dispatches requests to the assigned devices, but still requires loading and evicting models when the working set exceeds GPU memory. ServerlessLLM [29] introduces a loading-optimized checkpoint format and a multi-tier storage hierarchy that reduces cold-start latency by 10–200× compared to prior serverless frameworks, though models must still be fully loaded and unloaded for each switch. Most recently, Aegaeon [27] introduces token-level auto-scaling that dynamically loads and unloads models at fine granularity, reducing the GPU fleet needed to serve tens of models by 82% in a production deployment at Alibaba Cloud. Across all of these approaches, transferring model weights into and out of GPU memory remains the irreducible cost of multi-model serving, whether the source is disk, host DRAM, or a pre-staged CPU buffer.

A common assumption across the literature is that preemption carries a non-negligible cost, yet the actual magnitude of that cost, and its decomposition into constituent steps, has not been empirically isolated. Kim et al. [23] rely on simulated overhead rather than measured transfer times; FastServe [20] accounts for swap latency in its design but does not separate it from end-to-end throughput gains; FastSwitch [19] measures context-switch stalls but not the full unload–reload cycle. Similarly, none of the offloading systems cited above characterize the continuous relationship between the fraction of layers offloaded and the resulting throughput degradation. Our experiments address both gaps: Section IV sweeps the offloading ratio across models and hardware, and Section V decomposes the full preempt–resume cycle into its constituent steps.

III. Experimental Setup

A. Hardware

The offloading experiment is conducted on two separate servers that share the same CPU subsystem, an AMD Threadripper PRO 5995WX (64 cores, 512 GB DDR4) with a Samsung SSD 990 PRO 4TB, but differ in GPU hardware. The first server is equipped with two NVIDIA RTX 5000 Ada Generation GPUs (32 GB VRAM each, PCIe Gen 4 x16), and the second with two NVIDIA RTX A6000 GPUs (48 GB VRAM each, PCIe Gen 4 x16). The full offloading sweep is executed independently on each server to assess whether the qualitative behavior of layer offloading is consistent across GPU architectures while the CPU subsystem remains constant. The preemption experiment uses a single GPU on each server. Although both servers share the same PCIe Gen 4 x16 interface, the two GPUs differ in memory controller behavior and driver-level transfer scheduling, leading to measurable differences in both model reload times and KV cache transfer latencies. We therefore report preemption results separately for each platform to capture these hardware-dependent variations.

Table I
Models used. VRAM columns show the quantized (Q4) size for the offloading experiment and the FP16 size for the preemption experiment. Layer counts correspond to Ollama layers (transformer blocks + output layer) for offloading and to transformer layers for preemption.

Model	Params	Layers	Q4	FP16
Offloading experiment				
Llama 3 8B	8B	33	4.7 GB	—
Qwen3-32B	32B	65	20 GB	—
Llama 2 70B	70B	81	39 GB	—
Preemption experiment				
Qwen2.5-3B	3B	36	—	5.9 GB
Qwen3-8B	8B	36	—	15.6 GB
Qwen2.5-14B	14B	48	—	28.3 GB

B. Models and Software

Table I lists the models used across both experiments. The offloading experiment serves three models through Ollama v0.17.7 in their default quantized (Q4) formats: Llama 3 8B, Qwen3-32B, and Llama 2 70B. With Ollama we control the number of transformer layers placed on the GPU, sweeping through target GPU fractions from 0% (all CPU) to 100% at 10% increments, with additional fine-grained points at 92%, 94%, 96%, and 98% near full GPU residency. Because percentage targets are rounded to integer layer counts, duplicate configurations are removed, yielding 13 to 15 unique placements per model depending on the total layer count. The preemption experiment uses three models, Qwen2.5-3B, Qwen3-8B, and Qwen2.5-14B, loaded in FP16 directly via HuggingFace Transformers, which gives us programmatic access to the KV cache tensors needed for migration. KV state is migrated between GPU and host memory using explicit device-to-device tensor copies over PCIe. After each model unload, we force full GPU memory release and verify that allocated memory drops to driver-level residual before proceeding to the next step.

C. Workloads

The offloading experiment crosses 3 models × 13–15 unique GPU placements × 6 output lengths (50, 150, 300, 500, 1,000, and 5,000 tokens) × 3 repeats, totalling 774 inference calls per server. The model is unloaded and reloaded before every call to ensure a clean layer placement. The entire sweep is repeated on both servers to enable cross-hardware comparison.

The preemption experiment generates 7,000 tokens (Job A) and interrupts it at nine checkpoints ($N \in \{100, 200, 300, 500, 1{,}000, 2{,}000, 3{,}000, 4{,}000, 5{,}000\}$ tokens), running a 500-token Job B during each interruption. Four model pairings are tested: Qwen2.5-3B→Qwen3-8B, Qwen3-8B→Qwen3-8B, Qwen3-8B→Qwen2.5-14B, and Qwen2.5-14B→Qwen3-8B, each with 2 repeats and compared against an uninterrupted baseline. Job A uses greedy decoding throughout.

D. Metrics

For partial offloading, we report decode throughput (tokens per second, tok/s) as a function of the fraction of layers

Authorized licensed use limited to: Universita degli Studi di Roma La Sapienza. Downloaded on October 27,2026 at 15:13:24 UTC from IEEE Xplore. Restrictions apply.

placed on the GPU. This metric captures the steady-state performance of autoregressive generation, where decoding dominates end-to-end latency for sufficiently long outputs. By varying the number of offloaded layers, we quantify the trade-off between reduced GPU memory usage and the corresponding throughput degradation due to slower CPU execution and additional data movement. We also report normalized throughput relative to the fully GPU-resident baseline, which allows direct comparison across models and hardware platforms.

For preemption, we instrument the execution pipeline by measuring the duration of each step in the preempt-resume cycle: KV cache transfer from GPU to CPU, model unload, replacement model load and execution, original model reload, and KV cache restoration from CPU to GPU. We report the total preemption overhead, defined as the sum of these transfer and swap times, as well as its decomposition into model swap and KV transfer components. To contextualize the absolute cost, we quantify the overhead as a percentage of the uninterrupted baseline completion time. We further record the KV cache size in bytes at each checkpoint and measure effective PCIe bandwidth separately for each transfer direction (GPU-to-CPU and CPU-to-GPU).

IV. Layer-Wise CPU–GPU Offloading

The LLM inference can be performed on CPUs, GPUs, or hybrid configurations. In hybrid settings, the model parameters are partitioned across devices, with some layers stored in GPU memory and others in CPU memory. During inference, computation proceeds sequentially through the layers, and each layer is executed on the device where its parameters reside. When execution transitions between layers located on different devices, intermediate activations are transferred across the CPU–GPU interconnect to enable continuation of the forward pass.

Figure 1, 2, 3 report the throughput (tokens per second) depending on the fraction of layers that are executed on the GPU. The figures report different models of varying sizes under different prompt lengths on two different GPU hardware.

A first observation is that, across all model sizes, throughput exhibits a consistent inverse relationship with prompt length, with longer prompts yielding proportionally lower throughput. This behavior arises from the growth of the KV cache, whose memory footprint scales linearly with the total sequence length. As the prompt length increases, both memory bandwidth pressure and cache access costs rise, thereby limiting effective token generation throughput.

Additionally, the scaling behavior varies with model size. Smaller models show a sharp increase in throughput as the fraction of layers placed on the GPU approaches 100%, indicating a strong sensitivity to full GPU usage and a significant penalty from partial offloading. In contrast, larger models exhibit a more gradual, approximately linear improvement in throughput as GPU allocation increases, suggesting that incremental offloading yields smoother and more predictable performance gains. Although the GPUs differ in architectural characteristics and absolute performance, these qualitative trends are consistently observed across all evaluated devices.

Figure 4 presents inference throughput normalized to the 100% GPU baseline, where all model parameters reside on the GPU and computation is fully GPU-bound. As observed previously, larger models exhibit behavior that more closely approaches linear scaling with respect to the fraction of layers placed on the GPU.

A second notable trend is that normalized throughput is consistently higher on the RTX 5000 compared to the RTX A6000 under partial offloading. This effect arises from the smaller performance gap between the CPU and the RTX 5000. Because the RTX 5000 has lower peak throughput, transferring a portion of the computation to the CPU incurs a smaller relative performance penalty. In contrast, the RTX A6000's substantially higher compute capability amplifies the relative cost of CPU offloading, leading to greater degradation in normalized throughput.

In this scenario, the relative performance of the GPU and CPU must be jointly considered. The effectiveness of offloading depends not only on the absolute speed of the GPU, but also on its performance gap with respect to the CPU. Systems with high-end GPUs may experience disproportionately larger penalties from CPU offloading, whereas more balanced configurations, where CPU and GPU throughput are closer, can sustain higher normalized performance under hybrid execution. Consequently, scheduling policies should be hardware-aware, explicitly incorporating both GPU capability and CPU throughput to make optimal placement decisions.

V. Dynamic KV Cache Migration

In this section, we benchmark the cost of full job preemption: pausing a running inference request, evicting its model and cached state from the GPU to service a higher-priority job, and then resuming the original request from where it left off. Unlike partial offloading, which trades throughput for co-residency, preemption assumes exclusive GPU access for each job but pays a switching cost every time control changes hands. We restrict our analysis to a single preemption per job; the aggregate cost under repeated preemption depends on the scheduling policy and is left to future work. The central question is how large that one-time cost actually is, and which step of the preempt-resume cycle dominates the overhead.

We answer it with a controlled experiment on both hardware platforms described in Section III: the NVIDIA RTX 5000 Ada Generation (32 GB VRAM) and the NVIDIA RTX A6000 (48 GB VRAM), using a single GPU on each server. A long-running *Job A* generates 7,000 tokens using greedy decoding in FP16. At a designated checkpoint, after $N \in \{100, 200, 300, 500, 1000, 2000, 3000, 4000, 5000\}$ tokens, we preempt Job A by copying its KV cache tensors to host memory over PCIe, then fully unloading its weights via explicit deletion and GPU cache flushing. A second model, *Job B*, is loaded onto the free GPU and generates 500 tokens. Once Job B completes, its model is unloaded in the same manner, Job A's weights are reloaded from disk, the saved KV cache is restored from CPU to GPU, and generation resumes from the saved state to produce the remaining $7{,}000 - N$ tokens. Comparing Job A's total wall time against an uninterrupted baseline isolates the overhead introduced by preemption.

Four configurations cover different model-size pairings to isolate the effect of each model's footprint on preemption cost. The first run uses Qwen2.5-3B (5.9 GB in FP16) as the preempted model (Job A), providing a lightweight baseline. The second and third runs promote Qwen3-8B (15.6 GB) to Job A, with Job B being Qwen3-8B and Qwen2.5-14B (28.3 GB) respectively. The fourth run reverses the roles,

Authorized licensed use limited to: Universita degli Studi di Roma La Sapienza. Downloaded on October 27,2026 at 15:13:24 UTC from IEEE Xplore. Restrictions apply.

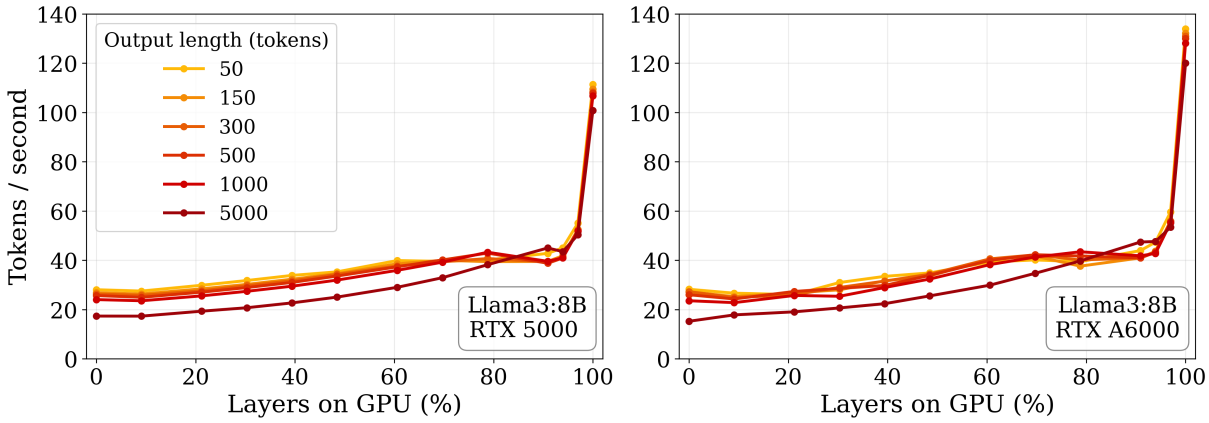


Figure 1. Throughput by the Layer allocation of GPU % for Llama3:8B model.

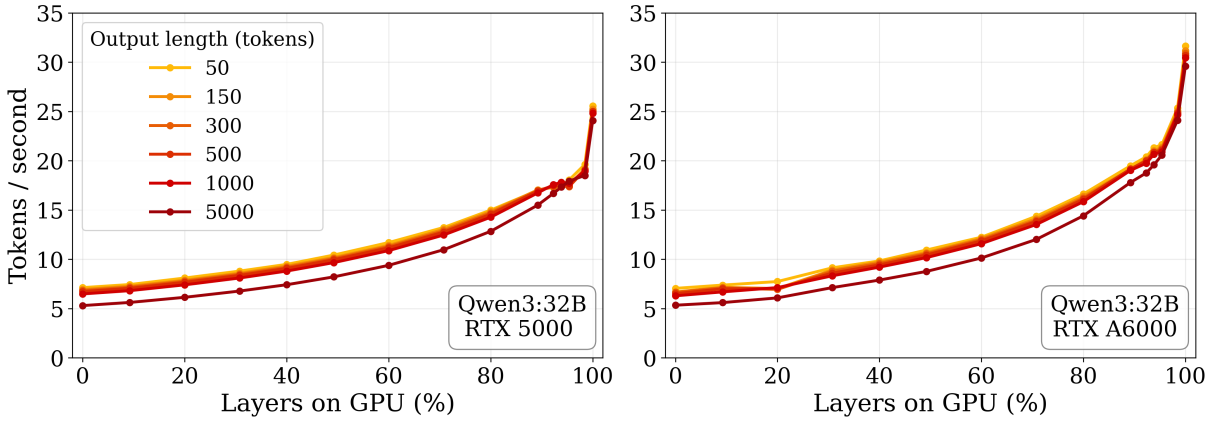


Figure 2. Throughput by the Layer allocation of GPU % for Qwen3:32B model.

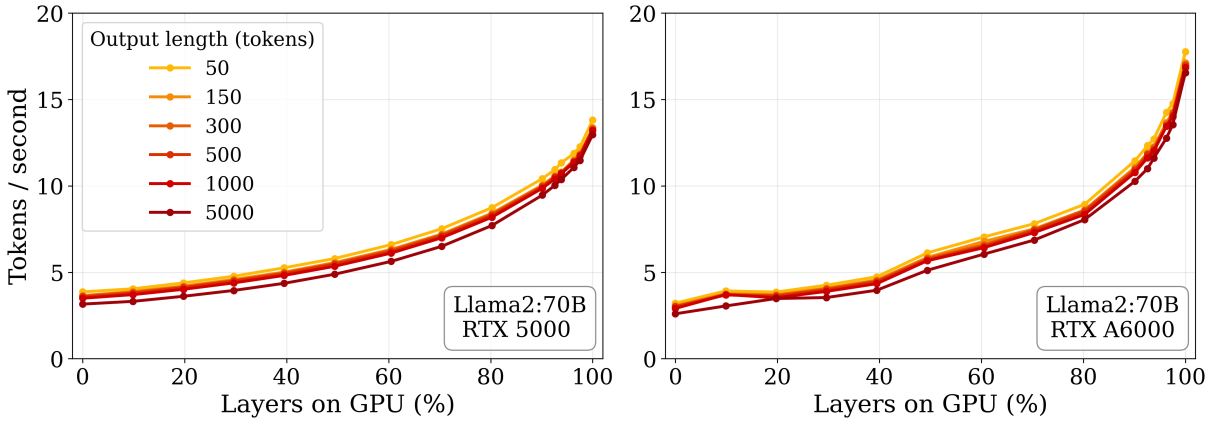
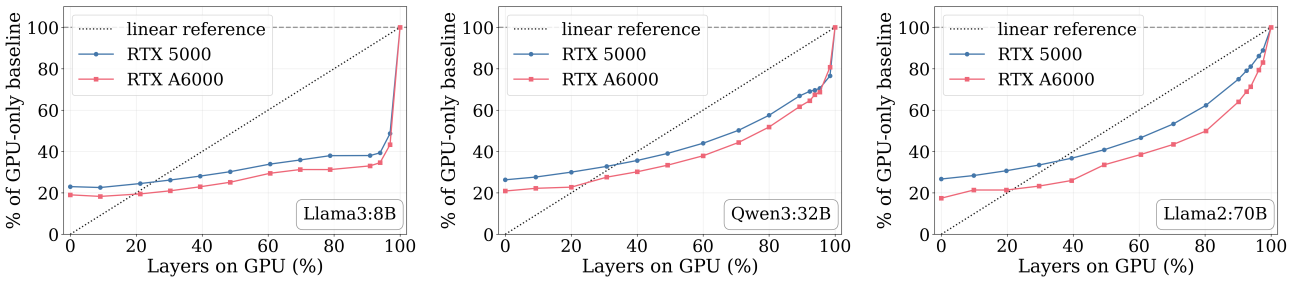


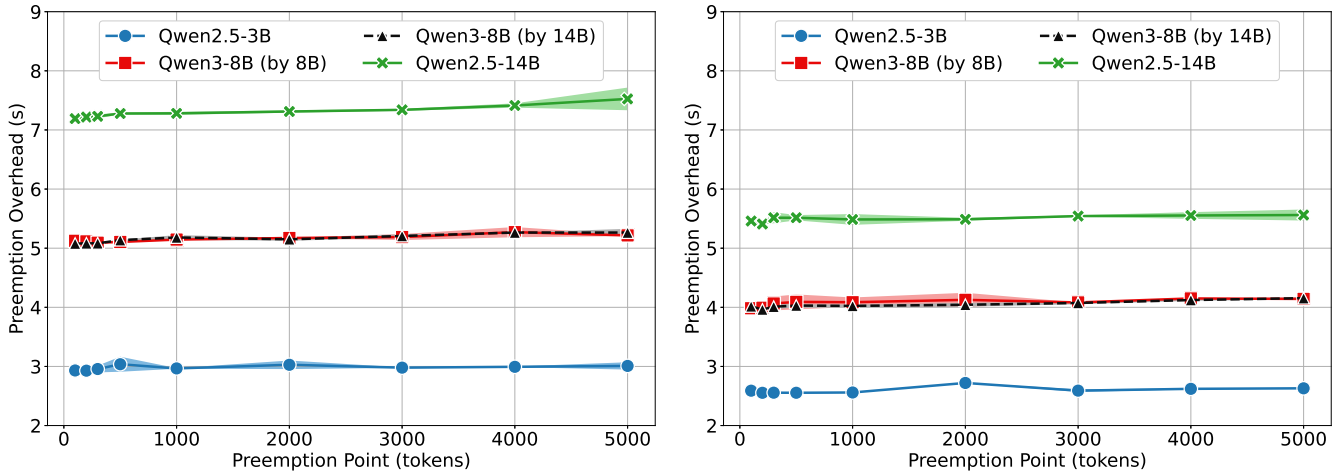
Figure 3. Throughput by the Layer allocation of GPU % for Llama2:70B model.



(a) Llama3:8B (b) Qwen3:32B (c) Llama2:70B

Figure 4. Normalized throughput.

Authorized licensed use limited to: Universita degli Studi di Roma La Sapienza. Downloaded on October 27,2026 at 15:13:24 UTC from IEEE Xplore. Restrictions apply.



(a) RTX 5000

(b) RTX A6000

Figure 5. Preemption overhead vs. preemption point for four model configurations. Shaded bands indicate one standard deviation.

preempting the larger Qwen2.5-14B with Qwen3-8B. Across these four pairings, the preempted model spans nearly a 5× range in memory footprint, from 5.9 to 28.3 GB, letting us disentangle the effect of the preempted model's size from that of the preempting model.

The sequence is deliberately conservative: we fully unload each model rather than attempting to keep both resident, which would only be possible when their combined footprint fits within VRAM. The results, therefore, represent a worst-case preemption cost for memory-constrained hardware.

Figure 5 plots the total preemption overhead as a function of the preemption point (after how many tokens are generated) for all four configurations on both GPUs. The central finding is immediate: every curve is effectively almost flat on both platforms, meaning that preemption overhead is constant regardless of how far the job has progressed at the time of interruption. On the RTX 5000, Qwen2.5-3B incurs approximately 3 s per preemption, both Qwen3-8B configurations settle around 5.1 s, and Qwen2.5-14B reaches approximately 7.3 s. On the RTX A6000, the same qualitative pattern holds but with consistently lower absolute overhead: Qwen2.5-3B drops to approximately 2.6 s, Qwen3-8B to around 4.1 s, and Qwen2.5-14B to roughly 5.7 s, reflecting faster disk-to-GPU transfer on that platform. In neither case do these values shift appreciably, whether the job is interrupted after 100 or 5,000 tokens. The shaded bands, which indicate one standard deviation across experimental repeats at each checkpoint, are narrow for the two smaller models and only slightly wider for Qwen2.5-14B, confirming that the result is not only flat but also robust across both hardware platforms. A second observation follows from comparing the two Qwen3-8B curves: the solid line (preempted by Qwen3-8B) and the dashed line (preempted by Qwen2.5-14B) are nearly indistinguishable on both GPUs, indicating that the size of the preempting model (Job B) has no measurable effect on the overhead experienced by the preempted job. What determines the cost is the preempted model's own weight footprint, not the identity of the job that displaces it.

Since the size of the preempting model has no measurable effect on overhead, we consolidate the two Qwen3-8B runs into a single average and report results per preempted model only. Table II breaks the total overhead into its four constituent steps, averaged across all checkpoints, for both GPUs. The breakdown reveals why overhead is constant: model reload dominates overwhelmingly. On the RTX 5000, reload accounts for 2.71 s, 4.70 s, and 6.70 s for Qwen2.5-3B, Qwen3-8B, and Qwen2.5-14B, respectively; on the RTX A6000, the corresponding values are 2.33 s, 3.62 s, and 5.12 s, consistently lower due to faster disk-to-GPU transfer on that platform. Adding the unload step, which is comparatively fast but still model-size-dependent, brings the model swap share to over 99% of total overhead on the RTX 5000 and above 98.5% on the RTX A6000 in every configuration. Unloading is not instantaneous because it involves three sequential operations: Python's cyclic garbage collector must traverse the entire module tree to detect and break reference cycles among the thousands of parameter tensors and sub-modules that constitute the model, the PyTorch caching allocator must then return all freed memory blocks to the CUDA driver via individual deallocation calls, and finally a device synchronization must complete before the released memory becomes available to the next model. Since both the number of objects to traverse and the number of cached blocks to free grow with model size, unload time scales from 0.26–0.28 s for Qwen2.5-3B to 0.53–0.54 s for Qwen2.5-14B across the two GPUs, with negligible difference between platforms as unloading is primarily a CPU-side operation. The KV cache transfers in both directions combined never exceed 1% of the total on the RTX 5000 and remain below 1.5% on the RTX A6000, contributing tens of milliseconds at most. This composition explains the flat curves in Figure 5: the dominant cost, reloading the model weights from disk into GPU memory, is a fixed function of the model's size and is entirely independent of how many tokens have been generated.

The last row of the table contextualizes the absolute cost in terms of end-to-end completion time. On the RTX 5000, Qwen2.5-3B completes 7,000 tokens in 147.5 s without interruption and in approximately 150 s with a single preemption, an overhead of roughly 2%. Qwen3-8B goes from a 244 s baseline to about 249 s, again around 2%. The largest model, Qwen2.5-14B, rises from 417 s to approximately 424 s, an overhead of 1.7%. On the RTX A6000, the relative overhead is even smaller: 1.72% for Qwen2.5-3B, 1.86% for Qwen3-8B, and 1.61% for Qwen2.5-14B, consistent with the lower

Authorized licensed use limited to: Universita degli Studi di Roma La Sapienza. Downloaded on October 27,2026 at 15:13:24 UTC from IEEE Xplore. Restrictions apply.

Table II
PREEMPTION OVERHEAD BREAKDOWN, AVERAGED ACROSS ALL CHECKPOINTS AND REPEATS.

	Qwen2.5-3B (5.9 GB)		**Qwen3-8B** (15.6 GB)		**Qwen2.5-14B** (28.3 GB)	
	RTX 5000	A6000	RTX 5000	A6000	RTX 5000	A6000
KV GPU → CPU (s)	0.0096	0.0125	0.0325	0.0387	0.0414	0.0439
Unload Model A (s)	0.26	0.28	0.41	0.39	0.54	0.53
Reload Model A (s)	2.71	2.33	4.70	3.62	6.70	5.12
KV CPU → GPU (s)	0.0076	0.0075	0.0211	0.0205	0.0276	0.0275
Total overhead (s)	2.98	2.62	5.16	4.06	7.31	5.73
Model swap (%)	99.4	99.2	99.0	98.5	99.1	98.8
KV transfer (%)	0.6	0.8	1.0	1.5	0.9	1.2
Overhead vs. baseline (%)	2.04	1.72	2.11	1.86	1.75	1.61

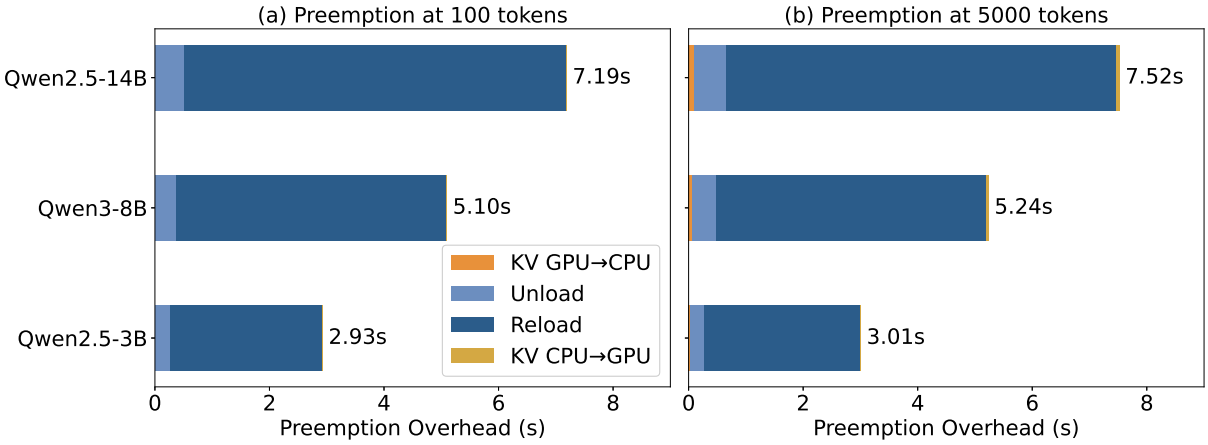


Figure 6. Overhead composition at the earliest and latest preemption points for different models. The GPU used is RTX 5000.

absolute preemption cost on that platform. In every case and on both GPUs, the preempted job finishes within seconds of the uninterrupted baseline, confirming that full preemption, even in this worst-case unload-reload scenario, is a lightweight operation whose cost is largely invisible at the job level. Figures 6 and 7 visualize this composition at the two extremes of the checkpoint range on the RTX 5000 and RTX A6000, respectively. At 100 tokens (left panels), the KV cache is just 6 MB for Qwen2.5-3B, 24 MB for Qwen3-8B, and 32 MB for Qwen2.5-14B, and the orange and yellow slivers representing KV transfers are almost invisible on both platforms. At 5,000 tokens (right panels), the KV cache has grown by roughly 30× to 178 MB, 713 MB, and 951 MB respectively, yet the bars are virtually identical in length. On the RTX 5000: 2.93 s versus 3.01 s for Qwen2.5-3B, 5.10 s versus 5.24 s for Qwen3-8B, and 7.19 s versus 7.52 s for Qwen2.5-14B. On the RTX A6000 the same pattern holds at lower absolute values: 2.64 s versus 2.67 s for Qwen2.5-3B, 4.00 s versus 4.15 s for Qwen3-8B, and 5.46 s versus 5.56 s for Qwen2.5-14B. The blue segment, model reload, visually overwhelms every other component in all panels across both GPUs, making it clear that even a 30× increase in the data that must be shuttled across PCIe produces no meaningful change in total overhead.

Although KV cache transfer contributes less than 1% of total preemption overhead on the RTX 5000 and below 1.5% on the RTX A6000, it is the only component whose cost grows with the length of the interrupted job. Characterizing this growth is important for two reasons: it reveals how efficiently the PCIe link is utilized during migration, and it establishes the regime in which KV transfer would begin to matter, for instance, on future hardware with faster model loading or for models serving very long contexts where caches reach tens of gigabytes. Table III reports the KV cache size and per-direction transfer time at each checkpoint for both GPUs.

The cache grows linearly with the number of generated tokens, at a rate that scales with model dimension: 0.035 MB/token for Qwen2.5-3B, 0.14 MB/token for Qwen3-8B, and 0.19 MB/token for Qwen2.5-14B. These rates are intrinsic to the model architecture and are therefore identical across both GPUs; only the transfer times differ. At the largest checkpoint (5,000 tokens), the cache reaches 178 MB, 713 MB, and 951 MB respectively, yet even the largest transfer completes in under 90 ms in the GPU-to-CPU direction and under 63 ms in the reverse direction on the RTX 5000. On the RTX A6000, GPU-to-CPU times are moderately higher, reaching up to 105 ms for Qwen2.5-14B at 5,000 tokens, while CPU-to-GPU times remain comparable at under 60 ms. The GPU-to-CPU path is consistently slower than CPU-to-GPU on both platforms because CUDA blocks device-to-host transfers until the copy has fully completed, whereas host-to-device transfers may return before the DMA operation finishes [30]. Effective bandwidth rises with transfer size as the fixed per-transfer overhead is amortized: at 5,000 tokens the GPU-to-CPU path reaches 10–12 GB/s and the CPU-to-GPU path reaches 13–16 GB/s on the RTX 5000, with the RTX A6000 achieving similar CPU-to-GPU rates but somewhat lower GPU-to-CPU throughput (9–10 GB/s), both well below the 31.5 GB/s theoretical peak of a PCIe Gen 4 x16 link due to the overhead of per-layer tensor copies through the PyTorch runtime. These numbers confirm that, even without optimized bulk-transfer implementations, KV cache migration is a fast operation whose cost is dwarfed by the time needed to reload

Authorized licensed use limited to: Universita degli Studi di Roma La Sapienza. Downloaded on October 27,2026 at 15:13:24 UTC from IEEE Xplore. Restrictions apply.

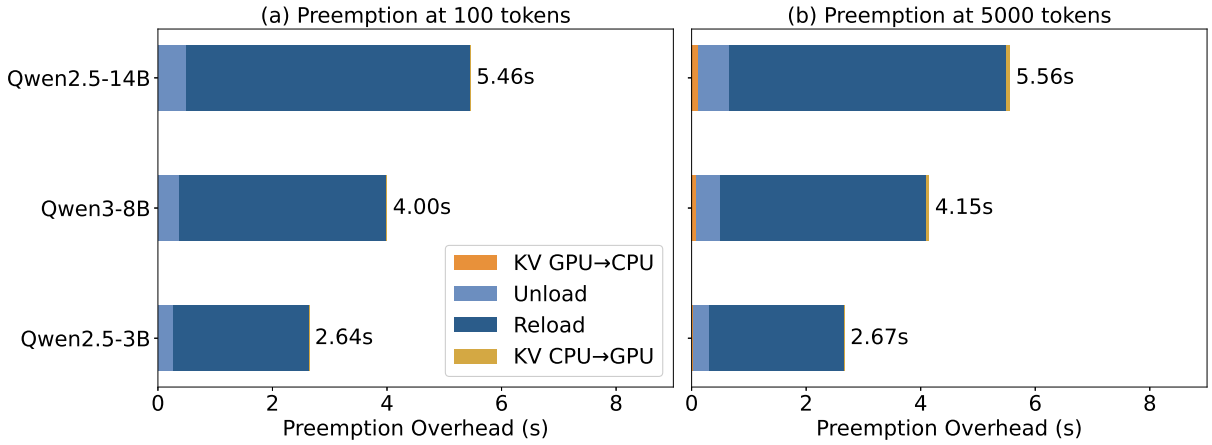


Figure 7. Overhead composition at the earliest and latest preemption points for different models. The GPU used is RTX A6000.

Table III
KV CACHE SIZE AND TRANSFER TIME ACROSS CHECKPOINTS.

Model	Tokens	KV	RTX 5000		RTX A6000	
		(MB)	GPU→CPU (ms)	CPU→GPU (ms)	GPU→CPU (ms)	CPU→GPU (ms)
Qwen2.5-3B	100	6.0	4.4	2.8	5.4	3.0
	500	20.1	5.7	4.8	5.8	5.0
	1000	37.7	8.2	7.9	7.3	8.0
	2000	72.8	11.8	8.6	11.4	8.5
	3000	108.0	12.9	10.0	18.0	9.9
	5000	178.3	18.0	14.4	31.3	14.1
Qwen3-8B	100	24.0	6.1	6.1	5.6	6.0
	500	80.3	12.0	9.0	11.2	8.9
	1000	150.6	18.7	13.5	28.5	13.7
	2000	291.2	33.0	22.3	50.4	22.8
	3000	431.9	45.7	30.4	65.7	30.5
	5000	713.1	67.8	45.9	90.3	45.4
Qwen2.5-14B	100	32.1	8.7	7.9	7.0	9.5
	500	107.1	16.0	12.9	14.9	12.7
	1000	200.8	29.7	18.6	30.7	18.6
	2000	388.3	45.8	30.1	54.4	29.5
	3000	575.8	62.1	41.1	70.5	39.7
	5000	950.8	86.7	62.5	104.9	59.8

model weights from disk on both platforms. Taken together, these results yield a simple rule of thumb for scheduling: the cost of preempting an LLM inference job can be modeled as a fixed penalty that depends only on the weight footprint of the preempted model, not on how much progress the job has made. On our PCIe Gen 4 hardware, this penalty ranges from roughly 3 s for a 3B-parameter model to 7 s for a 14B-parameter model on the RTX 5000, and from approximately 2.6 s to 5.7 s on the RTX A6000, adding no more than 2% to job completion time for long-running jobs such as the 7,000-token generation used in this experiment. KV cache migration, despite growing linearly with sequence length, remains negligible in comparison and would only become significant if model reload times were reduced by an order of magnitude. These findings simplify the design of preemption-aware schedulers, which can treat the swap cost as a known constant per model and hardware pair rather than a variable that depends on the work done.

VI. Discussion and Feature Set Identification

The experimental results highlight that scheduling LLM inference in heterogeneous CPU–GPU environments is fundamentally governed by a set of interacting factors that extend beyond simple load balancing. In this section, we distill these observations into a set of key features that future schedulers must explicitly consider when supporting partial offloading and preemption.

A first critical feature is the non-linear relationship between the fraction of layers placed on the GPU and the resulting throughput. Across all evaluated models, decode throughput does not scale linearly with the percentage of layers assigned to the GPU, deviating significantly from the linear reference. This effect is particularly pronounced for smaller models such as Llama3:8B or Qwen3:32B, where even moderate CPU offloading leads to substantial degradation in normalized performance, whereas larger models exhibit a flatter degradation curve and greater tolerance to heterogeneous execution. This implies that schedulers must incorporate model-specific sensitivity to offloading, rather than relying on proportional or uniform placement strategies.

A second feature is the strong dependence on model architecture, beyond raw parameter count. The results show clear differences between models of similar scale, such as Qwen and Llama, indicating that architectural choices influence how computation and memory are distributed across layers and therefore affect both throughput and offloading

Authorized licensed use limited to: Universita degli Studi di Roma La Sapienza. Downloaded on October 27,2026 at 15:13:24 UTC from IEEE Xplore. Restrictions apply.

efficiency. Consequently, scheduling decisions must be aware of model-specific execution profiles, rather than treating all models as interchangeable workloads.

A third feature is the interaction between offloading and sequence length. Throughput measurements across varying output lengths indicate that longer sequences amplify the impact of offloading, as decode-phase inefficiencies accumulate over time. While prefill costs are amortized, decode throughput becomes the dominant factor, and any slowdown introduced by CPU execution is repeatedly incurred. This suggests that schedulers must account for request characteristics, particularly expected output length, when deciding placement and resource allocation.

A fourth feature concerns the cost structure of preemption. The decomposition of preemption overhead reveals that the dominant component is not KV cache transfer, but rather model state reload, which accounts for over 99% of total overhead on the RTX 5000 and above 98.5% on the RTX A6000. This cost remains relatively stable across different preemption points, indicating that preemption overhead is largely insensitive to when it occurs during generation. However, the absolute overhead varies significantly across both models and hardware: from roughly 3 s for a 3B-parameter model to 7 s for a 14B-parameter model on the RTX 5000, and from 2.6 s to 5.7 s on the RTX A6000. This highlights that preemption is not a lightweight operation and must be used selectively, with awareness of both model-dependent and hardware-dependent costs.

A fifth feature is the role of KV cache size and data movement. Although KV transfer contributes less than 1.5% of total preemption overhead on both platforms, it is the only component that grows with the length of the interrupted job, and it becomes increasingly relevant for longer sequences where the cache scales linearly with token count. The measured PCIe bandwidth indicates that interconnect limitations can become a bottleneck, particularly in scenarios involving frequent preemption or aggressive offloading. Therefore, schedulers must consider both compute placement and data movement costs, especially under high concurrency or long-context workloads.

Finally, the results indicate that hardware characteristics significantly influence all of the above factors. Differences between GPUs (e.g., RTX 5000 vs. RTX A6000) affect not only the baseline throughput and the relative penalty of offloading, but also the absolute cost of preemption, with the RTX A6000 exhibiting lower reload times despite its larger VRAM capacity. This reinforces that scheduling policies must be hardware-aware, treating the swap cost as a known constant per model and hardware pair, and adapting decisions to the specific compute, memory, and I/O capabilities of the underlying devices.

Taken together, these observations suggest that effective scheduling for multi-model LLM inference requires a holistic approach that integrates model-aware, workload-aware, and hardware-aware features. Future schedulers should explicitly model (i) non-linear offloading sensitivity, (ii) model architecture and scale, (iii) sequence length and decode dynamics, (iv) preemption overhead decomposition, and (v) data movement constraints. Incorporating these features is essential to enable efficient preemption and hybrid CPU–GPU execution in realistic, heterogeneous deployment scenarios.

VII. Conclusions

In this paper, we presented a fine-grained empirical study of CPU–GPU layer offloading and job-level preemption for LLM inference, deriving a set of design features for schedulers targeting multi-model LLM deployments. Our measurements expose two regimes that current single-model schedulers do not capture. First, decode throughput degrades non-linearly with partial offloading, and the shape of this degradation depends on both the model and the GPU, so a single offloading curve cannot drive placement decisions. Second, the cost of a full preempt-resume cycle is effectively almost constant regardless of how far the interrupted job has progressed, since model reload from disk dominates the overhead while KV cache migration remains a minor contribution even at long sequence lengths. Together, these results allow preemption-aware schedulers to treat the swap cost as a known, fixed penalty per model and hardware pair, and the features distilled in Section VI translate these regimes into concrete inputs that a multi-model scheduler can act on.

As future work, the offloading and preemption characterization should be extended to continuous-batching scenarios with concurrent requests, since all experiments in this study use single-request execution. The preemption analysis should also be generalized beyond a single interruption per job, as the aggregate cost under repeated preemptions will depend on the scheduling policy and may exhibit different trade-offs. A workload-level study is needed to estimate how often a typical inference job would need to be preempted under realistic arrival patterns, and to compare the resulting performance against fully non-preemptive policies, in order to determine whether preemption yields a net benefit or a net loss on average.

Acknowledgment

This work was supported by the European Union - Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, CUP B53C22004050001, partnership on “Telecommunications of the Future” (PE00000001 - program “RESTART”)

References

[1] 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.

[2] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, “Flexgen: High-throughput generative inference of large language models with a single gpu,” in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 094–31 116.

[3] H. Kim, N. Wang, Q. Xia, J. Huang, A. Yazdanbakhsh, and N. S. Kim, “Lia: A single-gpu llm inference acceleration with cooperative amx-enabled cpu-gpu computation and cxl offloading,” in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 2025, pp. 544–558.

[4] Y. Song, Z. Mi, H. Xie, and H. Chen, “Powerinfer: Fast large language model serving with a consumer-grade gpu,” in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024, pp. 590–606.

[5] K. Kamahori, T. Tang, Y. Gu, K. Zhu, and B. Kasikci, “Fiddler: Cpu-gpu orchestration for fast inference of mixture-of-experts models,” 2024.

[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] X. Jiang, Y. Zhou, S. Cao, I. Stoica, and M. Yu, “Neo: Saving gpu memory crisis with cpu offloading for online llm inference,” vol. 7, 2025.

Authorized licensed use limited to: Universita degli Studi di Roma La Sapienza. Downloaded on October 27,2026 at 15:13:24 UTC from IEEE Xplore. Restrictions apply.

[8] J. Wu, J. Ren, S. Yang, K. Parasyris, G. Georgakoudis, I. Laguna, and D. Li, "Lm-offload: Performance model-guided generative inference of large language models with parallelism control," IEEE, pp. 840–849, 2025.

[9] C. Hooper, S. Kim, H. Mohammadzadeh, M. W. Mahoney, Y. S. Shao, K. Keutzer, and A. Gholami, "Kvquant: Towards 10 million context length llm inference with kv cache quantization," *Advances in Neural Information Processing Systems*, vol. 37, pp. 1270–1303, 2024.

[10] 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.

[11] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, "Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills," *arXiv preprint arXiv:2308.16369*, 2023.

[12] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in llm inference with sarathi-serve," in *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, 2024, pp. 117–134.

[13] 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.

[14] 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.

[15] F. Strati, S. McAllister, A. Phanishayee, J. Tarnawski, and A. Klimovic, "Déjàvu: Kv-cache streaming for fast, fault-tolerant generative llm serving," in *Proceedings of the 41st International Conference on Machine Learning*, 2024, pp. 46 745–46 771.

[16] W. Lee, J. Lee, J. Seo, and J. Sim, "Infinigen: Efficient generative inference of large language models with dynamic kv cache management," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 155–172.

[17] X. Pan, E. Li, Q. Li, S. Liang, Y. Shan, K. Zhou, Y. Luo, X. Wang, and J. Zhang, "Instattention: in-storage attention offloading for cost-effective long-context llm inference," pp. 1510–1525, 2025.

[18] B. Sun, Z. Huang, H. Zhao, W. Xiao, X. Zhang, Y. Li, and W. Lin, "Llumnix: Dynamic scheduling for large language model serving," in *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, 2024, pp. 173–191.

[19] A. Shen, Z. Li, and M. Gao, "Fastswitch: Optimizing context switching efficiency in fairness-aware large language model serving," *arXiv preprint arXiv:2411.18424*, 2024.

[20] B. Wu, Y. Zhong, Z. Zhang, S. Liu, F. Liu, Y. Sun, G. Huang, X. Liu, and X. Jin, "Fast distributed inference serving for large language models," *arXiv preprint arXiv:2305.05920*, 2023.

[21] Y. Jin, C.-F. Wu, D. Brooks, and G.-Y. Wei, "s^3: Increasing GPU utilization during generative inference for higher throughput," vol. 36, 2023, pp. 18 015–18 027.

[22] Z. Zheng, X. Ren, F. Xue, Y. Luo, X. Jiang, and Y. You, "Response length perception and sequence scheduling: An llm-empowered llm inference pipeline," *Advances in Neural Information Processing Systems*, vol. 36, pp. 65 517–65 530, 2023.

[23] K.-M. Kim, K. Hong, C. Gulcehre, and A. Ailamaki, "The effect of scheduling and preemption on the efficiency of llm inference serving," 2024.

[24] X. Miao, C. Shi, J. Duan, X. Xi, D. Lin, B. Cui, and Z. Jia, "Spotserve: Serving generative large language models on preemptible instances," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 1112–1127.

[25] M. Heisler, Z. Yousefijamarani, X. Wang, Q. Wang, G. Shi, H. Sadri, T. Yu, Y. Li, H. Li, G. Singh *et al.*, "Llm inference scheduling: A survey of techniques, frameworks, and trade-offs," *Authorea Preprints*, 2025.

[26] J. Duan, R. Lu, H. Duanmu, X. Li, X. Zhang, D. Lin, I. Stoica, and H. Zhang, "Muxserve: flexible spatial-temporal multiplexing for multiple llm serving," 2024.

[27] Y. Xiang, X. Li, K. Qian, Y. Yang, D. Zhu, W. Yu, E. Zhai, X. Liu, X. Jin, and J. Zhou, "Aegaeon: Effective gpu pooling for concurrent llm serving on the market," in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, 2025, pp. 1030–1045.

[28] S. Yu, J. Xing, Y. Qiao, M. Ma, Y. Li, Y. Wang, S. Yang, Z. Xie, S. Cao, K. Bao *et al.*, "Prism: Unleashing gpu sharing for cost-efficient multi-llm serving," *arXiv preprint arXiv:2505.04021*, 2025.

[29] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "Serverlessllm: Low-latency serverless inference for large language models," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 135–153.

[30] NVIDIA Corporation, *CUDA C++ Programming Guide – API Synchronization Behavior*, 2025, accessed: 2025. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#api-synchronization-behavior.

Authorized licensed use limited to: Universita degli Studi di Roma La Sapienza. Downloaded on October 27,2026 at 15:13:24 UTC from IEEE Xplore. Restrictions apply.