AegisKV: Runtime Headroom Control for Online LLM Serving via Adaptive Attention Offloading

Kihyun Kim
Sogang University
Seoul, Republic of Korea
kion777@sogang.ac.kr

Seong Jang
Sogang University
Seoul, Republic of Korea
jseong@sogang.ac.kr

Min Heo
Sogang University
Seoul, Republic of Korea
heomin02@sogang.ac.kr

Jinwoo Kim
Sogang University
Seoul, Republic of Korea
jinwookim@sogang.ac.kr

James J. Kim
Soteria Inc.
Seoul, Republic of Korea
jkim@soteria-sys.com

Youngjae Kim*
Sogang University
Seoul, Republic of Korea
youkim@sogang.ac.kr

Abstract

This paper presents AegisKV, a runtime-adaptive attention offloading system for heterogeneous GPU environments. AegisKV dynamically offloads decode attention and KV cache at the request level based on runtime KV pressure and auxiliary GPU capability. AegisKV treats attention offloading not simply as KV-capacity expansion, but as a runtime KV headroom control mechanism for continuous batching in online LLM serving. To realize this design, AegisKV integrates (i) a pressure-aware request offloading scheduler, (ii) heterogeneous KV-cache management across serving and auxiliary GPUs, and (iii) a cross-device execution pipeline that overlaps KV transfer and decode execution. By selectively offloading only necessary requests, AegisKV mitigates preemption and recomputation while avoiding unnecessary auxiliary-path overhead. We implement AegisKV on top of vLLM and evaluate it using realistic online serving workloads on heterogeneous GPU configurations. On Llama-2-13B, AegisKV achieves up to 1.23× lower normalized token latency than baseline vLLM and 1.80× lower than static full offloading. Furthermore, on Qwen2.5-72B with MPS-based optimization, AegisKV achieves 15.07× lower TTFT and delays the onset of latency saturation by up to 1.43×.

CCS Concepts

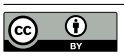
• **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Computing methodologies** → *Natural language generation.*

Keywords

Online LLM Serving, LLM Inference, Attention Offloading, KV Cache Management, Continuous Batching, Heterogeneous GPU Serving

ACM Reference Format:
Kihyun Kim, Seong Jang, Min Heo, Jinwoo Kim, James J. Kim, and Youngjae Kim. 2026. AegisKV: Runtime Headroom Control for Online LLM Serving via Adaptive Attention Offloading. In *Proceedings of the 35th ACM International Conference on Information and Knowledge Management (CIKM '26), November 07–11, 2026, Rome, Italy.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3799682.3841146

*Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.
CIKM '26, Rome, Italy
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2539-5/2026/11
https://doi.org/10.1145/3799682.3841146

1 Introduction

Large Language Models (LLMs) have been widely deployed in interactive online services such as conversational agents, coding assistants, and agentic workflows [5, 15, 27]. These services require LLM serving systems to handle many concurrent requests while maintaining low serving latency. Modern serving engines therefore use continuous batching, which admits new requests into an ongoing batch at iteration boundaries to improve GPU utilization and throughput [1, 14, 28].

A key limitation of continuous batching is KV cache capacity. In Transformer-based LLMs, the KV cache stores the key and value tensors used by attention layers to reuse previous-token context during generation [25]. As more requests are served concurrently, their aggregate KV cache footprint can exceed the available capacity. When KV capacity is exhausted, runtimes such as vLLM preempt requests and later recover their states through recomputation [14]. This wastes completed generation work, consumes additional GPU time, and delays new request admission, directly increasing online latency such as time-to-first-token (TTFT).

A straightforward way to increase serving capacity is model parallelism, which distributes model parameters and KV cache across multiple GPUs [22]. However, this expands KV capacity only by scaling the full-model execution structure. For example, tensor parallelism relies on coarse-grained partitioning such as attention-head-level splitting, so adding GPUs also changes how the model is executed. This makes model parallelism a costly and indirect way to relieve KV pressure when the primary bottleneck is KV cache capacity rather than model computation.

Attention offloading (AO) takes a more targeted approach, moving KV cache storage and the corresponding attention computation outside the serving GPU [4, 7, 9, 10, 13, 17, 18]. Unlike model parallelism, AO does not require the auxiliary device to host the full model: the serving GPU continues executing the main model computation, while the auxiliary device stores KV cache and computes attention over that cache. CPU-based AO is limited by host-side bandwidth and latency [18], while accelerator-based AO requires dedicated hardware support [13, 18]. In contrast, GPU-based AO

provides a practical approach for existing GPU clusters by expanding effective KV capacity with an auxiliary GPU. A representative GPU-based AO design is Lamina [4], which adopts a *static full AO* policy that routes every request through the auxiliary path regardless of runtime KV pressure or auxiliary GPU capability.

Motivation. Lamina's effectiveness is not determined by capacity expansion alone. Static full AO can reduce serving-side KV usage, but it also introduces a trade-off between KV-pressure relief and auxiliary-path overhead. Our measurements show that this policy does not consistently match online serving workloads (§3.1). When KV pressure is low, routing all requests through the auxiliary GPU introduces unnecessary transfer and synchronization overhead, increasing TTFT by up to 33% over the baseline. Conversely, under high KV pressure, if the auxiliary GPU is less capable than the serving GPU, the auxiliary path becomes a new bottleneck, increasing TTFT by up to 157% over the baseline. These results indicate that GPU-based AO should not be treated as an always-on execution path, but rather as an *adaptive offloading mechanism* whose decisions respond to runtime KV pressure and auxiliary GPU capability.

Approach. In this paper, we present AegisKV, a pressure-reactive GPU-based AO system for online LLM serving. It augments an existing LLM serving deployment with a heterogeneous auxiliary GPU dedicated to KV-cache storage and decode attention. Unlike static full AO, AegisKV offloads only selected requests based on runtime KV pressure and auxiliary-path cost, thereby avoiding auxiliary-side bottlenecks (§3.1). This expands serving-side KV capacity, reduces preemption-inducing KV pressure, and improves TTFT and normalized token latency. To realize this design, AegisKV integrates three mechanisms. (i) **AegisKV-Control** dynamically determines when and which requests to offload based on serving-side KV pressure and auxiliary-path conditions (§4.2). (ii) **AegisKV-Store** manages heterogeneous KV-cache placement, ownership, and migration across the serving and auxiliary GPUs (§4.3). (iii) **AegisKV-Flow** coordinates the execution of local and offloaded requests within the same continuous batching loop, overlapping cross-device data transfer with computation (§4.4).

Contributions. This paper makes the following contributions.

- We show that GPU-based attention offloading in online LLM serving is not merely a KV-capacity expansion mechanism, but a runtime KV-headroom control problem under continuous batching. We characterize how transient KV pressure causes preemption-induced progress loss and why static full AO fails to balance serving-side headroom recovery against decode-side overhead (§3). To the best of our knowledge, this is the first systematic analysis of attention offloading in vLLM under runtime KV-pressure dynamics and varying KV-headroom conditions.
- We design and implement AegisKV, a pressure-reactive request-level attention offloading system for heterogeneous GPUs. It determines both when to activate offloading and which requests to offload from runtime KV pressure, request KV footprint, and auxiliary-path slowdown, while preserving continuous batching and block-allocation semantics in vLLM (§4).
- We extensively evaluate AegisKV across heterogeneous GPU configurations and online serving workloads. It mitigates latency degradation under KV pressure, improving TTFT and normalized token latency over state-of-the-art static AO [4] and baseline vLLM, and improves serving cost efficiency through low-cost auxiliary GPUs (§5).

2 Background and KV-Pressure Dynamics

KV cache is a key runtime state in LLM serving, whose memory demand dynamically changes with request concurrency and sequence progress. This section describes how such KV pressure interacts with GPU memory headroom and the serving runtime.

2.1 KV Cache and Memory Headroom

LLMs generate tokens sequentially through Transformer-based autoregressive inference [25]. At each step, the model attends the query of the current token to the keys and values of all previous tokens. Recomputing these keys and values at every step would incur substantial redundant computation, so serving systems retain them in GPU memory as a KV cache and append new entries as generation proceeds.

LLM inference is generally divided into two phases: *prefill* and *decode*. The prefill phase processes the entire input prompt in parallel and generates the initial KV cache, and it has relatively high arithmetic intensity [12]. In contrast, the decode phase generates output tokens one at a time; at every step, it reads the accumulated KV cache to perform attention and appends a new KV entry [12, 19]. In this phase, decode attention uses almost no model parameters, but must repeatedly access a KV cache that grows in proportion to sequence length. Therefore, the decode phase becomes sensitive to GPU memory capacity and memory bandwidth under high concurrency or long-context settings [26].

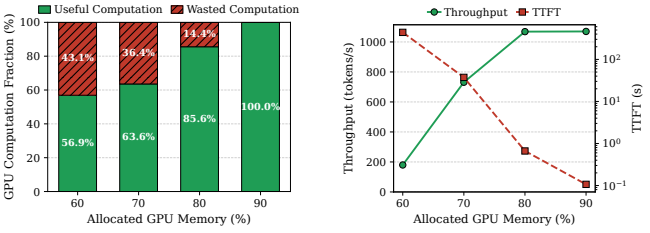
GPU Memory Headroom. The footprint of the KV cache increases with model size, context length, and the number of concurrently active requests. In particular, unlike model weights, the KV cache is maintained as per-request runtime state, so long-context serving and high request concurrency can quickly consume the remaining GPU memory headroom. As a result, limited KV headroom directly affects both the number of active requests that the serving runtime can maintain and the decode progress of each request.

2.2 vLLM Runtime under KV Cache Pressure

vLLM [14] is a representative open-source online LLM serving engine that provides high serving throughput and GPU memory utilization through continuous batching, PagedAttention-based KV-cache management, and preemption recovery via recomputation.

Continuous Batching. Online LLM serving systems use continuous batching to process requests with different arrival times and output lengths at high throughput [1, 14, 28, 30]. Unlike static batching, which stalls until every request in a batch completes, continuous batching updates the running batch at every scheduling step. Completed requests are removed from the batch, while waiting requests are admitted as resources become available. As a result, the active request set continuously changes at runtime according to request arrivals, completions, and decode progress.

PagedAttention and KV Block Management. This dynamic set requires the serving system to manage each request's KV cache at



(a) Wasted computation ratio. **(b) Throughput and TTFT.**

Figure 1: Impact of reduced KV capacity under a fixed load.

runtime. To this end, modern LLM serving systems use PagedAttention to construct a KV-block pool in GPU memory and allocate each request's KV cache in fixed-size blocks [14]. Each request is allocated initial KV blocks during the prefill phase, and as decoding progresses, additional blocks are allocated according to the increase in sequence length. When a request completes, the KV blocks it occupied are returned to the pool.

KV demand grows at runtime as running requests extend their sequences and continuous batching admits new requests. Because output lengths are not known in advance, the runtime cannot predict when blocks will be returned. When available KV blocks become insufficient, the serving runtime interrupts some requests and reclaims their KV blocks to prevent out-of-memory. This is called *preemption*.

For a preempted request to be executed again, the KV state up to the point of interruption must be recovered. Representative recovery methods include a swap-based method, which moves the KV cache to host memory and later reloads it [11], and a recomputation-based method, which reconstructs the KV state by re-executing the previous execution progress. However, due to the limitation of host-device I/O bandwidth, representative serving systems such as vLLM use recomputation-based recovery as the default method [14]. In this case, when a preempted request is rescheduled, it must re-execute the previous prefill and decode progress. Therefore, in modern LLM serving, KV-cache pressure appears not as a simple memory allocation problem, but as a runtime issue that directly affects request scheduling and execution overhead [23].

Impact of KV Capacity on Online Serving Stability. Because GPU memory headroom varies depending on model residency and deployment configuration, the KV capacity available to the serving runtime can differ even under the same GPU environment. To analyze the effect of this capacity constraint on online serving behavior, we run Llama-2-13B on a single L40S GPU and conduct experiments under a fixed 3 requests-per-second (RPS) Poisson arrival workload and 128/256 input/output lengths. To progressively reduce the GPU memory fraction allocated to the serving runtime, we adjust GPU utilization in vLLM and limit the memory headroom available to the KV-block pool.

As shown in Figure 1a, as the GPU memory fraction decreases, the fraction of computation wasted due to recomputation-based preemption increases. In the most constrained setting, 43.1% of the total GPU computation is used for preemption-induced recomputation. As a result, as shown in Figure 1b, throughput decreases and time-to-first-token (TTFT) increases. This shows that limited serving-side KV capacity can increase preemption-induced recomputation and consequently act as a major runtime bottleneck that degrades online serving stability.

3 Motivation and Preliminary Results

To mitigate runtime bottlenecks caused by limited serving-side KV capacity, recent studies have proposed *attention offloading (AO)* [4, 7–10, 18]. AO alleviates serving-side KV pressure by decoupling decode-stage attention and its associated KV cache from the serving GPU and executing them on an auxiliary device.

This decoupling is practical because decode attention is largely orthogonal to model-parameter computation: it primarily operates on the accumulated KV cache and is dominated by KV-cache accesses. As a result, decode attention can be treated as a memory-bound operation that can run independently on a device with sufficient memory bandwidth. In GPU-based AO, at each decode iteration, the serving GPU transfers the query and newly generated key/value tensors for the current token to the auxiliary GPU. The auxiliary GPU appends the new key/value tensors to its resident KV cache, computes attention over the accumulated KV cache, and returns the attention output to the serving GPU so that the remaining forward computation can continue.

Lamina [4] is a representative model-attention disaggregation system that uses the auxiliary GPU as an attention-specialized device for active requests rather than a full model worker, thereby reducing KV-cache residency on the serving GPU and allowing a larger active batch under high-concurrency workloads.

3.1 Why Static Attention Offloading Fails

We evaluate *Lamina*, our implementation of a representative *Static Full AO* baseline that follows Lamina's model-attention disaggregation policy. Static Full AO offloads every request's decode attention and KV cache to the auxiliary GPU. Although this policy increases the KV capacity available to the serving system, its effectiveness is not determined by KV-capacity expansion alone. Online serving workloads exhibit time-varying KV pressure, request concurrency, and sequence-length distributions, while the auxiliary GPU may differ from the serving GPU in compute capability and memory bandwidth. Therefore, Static Full AO can act either as useful KV relief or as avoidable auxiliary-path overhead, depending on the workload and hardware configuration.

To quantify this behavior, we run Llama-2-13B on a single L40S serving GPU. The baseline uses no attention offloading, while Static Full AO is evaluated with either an auxiliary L40S (*AO-L40S*) or an RTX 4000 Ada GPU (*AO-RTX4000*). Requests arrive according to a Poisson process with fixed 512/256 input/output token lengths. Table 1 reports *TTFT* (time-to-first-token), *TPOT* (time-per-output-token), and *output throughput*.

Problem 1: Unnecessary overhead at low pressure. At 0.5 RPS, where KV pressure is low, baseline serves requests without preemption, and the total KV working set remains within the capacity of a single serving GPU. In this regime, AO is not required for capacity relief. However, Static Full AO routes every request to the auxiliary GPU regardless of the runtime serving state, incurring KV transfer and attention round-trip overhead at every decode iteration.

Table 1: Runtime impact of static full attention offloading.

RPS	Configuration	TTFT (s)	TPOT (ms)	Output (tok/s)
0.5	Baseline	0.12	44.0	125.9
	AO-L40S	0.15 (+25%)	49.1 (+11%)	125.7 (-0.2%)
	AO-RTX4000	0.16 (+33%)	60.3 (+37%)	125.6 (-0.3%)
2.0	Baseline	18.72	72.2	386.7
	AO-L40S	0.20 (-99%)	92.2 (+28%)	482.0 (+25%)
	AO-RTX4000	48.19 (+157%)	124.7 (+73%)	275.2 (-29%)

As a result, TTFT increases by 25% with AO-L40S and 33% with AO-RTX4000 compared to baseline, while TPOT also increases by 11%–37%. In contrast, output throughput remains nearly unchanged from baseline. This shows that, under low KV pressure, Static Full AO provides little KV-capacity benefit but still adds communication and synchronization overheads. Thus, when KV pressure is sufficiently low, keeping offloading always enabled introduces pure latency overhead.

Problem 2: Auxiliary-side bottleneck at high pressure. At 2.0 RPS, where KV pressure is high, baseline suffers from frequent preemption and recomputation, increasing TTFT to 18.72 s and limiting output throughput to 386.7 tokens/s. In this regime, AO-L40S effectively relieves serving-side KV pressure because the auxiliary GPU provides a processing capability comparable to the serving GPU: it reduces TTFT to 0.20 s and improves output throughput by 25% over baseline. However, TPOT still increases by 28%, reflecting the cross-device transfer, auxiliary attention execution, and synchronization overheads introduced by the offloading path.

In contrast, AO-RTX4000 cannot keep up with the offloaded attention workload. The bottleneck therefore shifts from serving-side KV capacity to the offloading path itself, increasing TTFT to 48.19 s, which is 157% worse than baseline, and reducing output throughput by 29%. This result shows that, under Static Full AO, insufficient auxiliary-side capability can turn AO from a KV-pressure mitigation mechanism into a new runtime bottleneck.

The above analysis suggests that the key challenge in online LLM serving is not merely insufficient KV capacity, but runtime KV-headroom management. Under continuous batching, KV-cache demand continuously changes over time as requests arrive, progress through decoding, and complete generation. Therefore, the system must not only provide additional KV memory, but also maintain enough runtime KV headroom for the serving GPU to admit and sustain active requests under transient KV pressure.

When such headroom is insufficient, the serving runtime triggers request preemption to free KV-cache blocks. Since the discarded KV cache must later be reconstructed, preemption introduces additional prefill and decode computation. This preemption-induced recomputation increases TTFT and normalized token latency, delays the admission of new requests, and ultimately destabilizes online serving performance.

3.2 Challenges in Adaptive GPU-Based AO

Effective AO requires runtime adaptation: it should preserve KV headroom when needed rather than statically expand KV capacity.

Realizing this adaptive behavior introduces three key challenges.

Challenge 1: Pressure-aware request-level offloading. Adaptive AO requires pressure-aware request-level offloading that accounts for both runtime KV pressure and auxiliary-GPU capability. As shown above, AO can introduce unnecessary overhead when KV pressure is low, and can become a new bottleneck when the auxiliary GPU cannot keep up with the offloaded attention workload. Therefore, the serving runtime must decide whether to activate AO based on the current serving state, and apply offloading selectively to only a subset of requests rather than all requests.

Challenge 2: Hybrid decode execution across GPUs. Request-level AO creates a hybrid decode execution path across the serving GPU and the auxiliary GPU. The serving GPU continues to execute model-weight-dependent operators, while the auxiliary GPU computes decode attention over the offloaded KV cache. The runtime must therefore coordinate tensor transfers and auxiliary attention execution within each decode iteration, while minimizing decode-latency inflation caused by cross-device coordination.

Challenge 3: Serving-runtime integration. AO is not a simple external kernel invocation; it changes the serving runtime itself. Offloaded requests must remain consistent with continuous batching, PagedAttention-based KV-block allocation, and admission and preemption policies. Since the serving GPU and auxiliary GPU maintain separate KV-cache capacities, the runtime must track per-request KV-cache placement and ownership across GPUs, and provide globally consistent scheduling and memory management. Without such integration, AO can violate the runtime's assumptions about where KV blocks reside, which requests are admissible, and which requests can be safely preempted.

4 AegisKV Design and Runtime Mechanisms

AegisKV is designed to maintain sufficient KV headroom on the model-serving GPU by adaptively offloading selected requests to an auxiliary attention GPU. To achieve this, AegisKV jointly coordinates when and which requests to offload, where their KV state is maintained, and how offloaded attention is integrated into the serving execution path.

4.1 Overview

Figure 2 illustrates the architecture of AegisKV and its request execution flow. AegisKV augments the existing model-serving GPU (M-GPU), which holds model weights and executes the standard transformer forward pipeline, with an attention GPU (Attn-GPU) that is used for runtime KV-headroom control. When runtime KV headroom becomes insufficient, AegisKV redirects selected requests to the AO path, where their KV cache and decode attention are moved to the Attn-GPU. This relieves serving-side KV pressure and restores available KV headroom on the M-GPU.

The request execution in AegisKV proceeds as follows. First, the scheduler monitors runtime KV pressure and selects a subset of requests entering the running batch for the AO path (①). The M-GPU then performs prefill for both local-path and AO-path requests (②). For AO-path requests, the KV cache generated during prefill is migrated to the Attn-GPU; once the migration completes, the corresponding KV blocks on the M-GPU are released (③). During decoding, local-path requests execute on the M-GPU as in a standard serving engine. In contrast, AO-path requests transfer the query and newly generated key/value tensors for the current token to the Attn-GPU at each decode iteration (④). The Attn-GPU performs decode attention over its resident KV cache and returns the attention

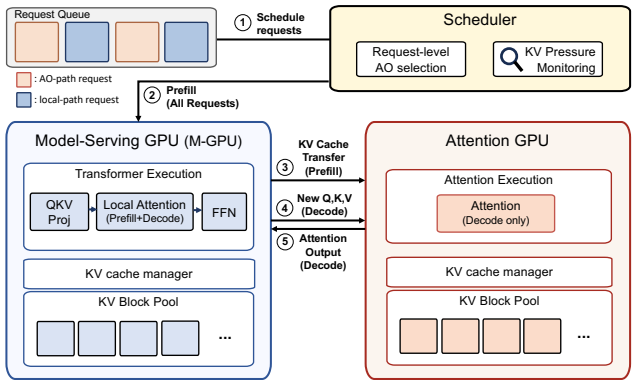


Figure 2: Architecture Overview of AegisKV.
AegisKV augments a model-serving GPU (M-GPU) with an Attn-GPU for selected offloaded-path requests. ①-⑤ indicate the per-request execution and KV-migration flow.

output to the M-GPU (⑤). The M-GPU then uses the returned attention output to continue the remaining transformer computation, including output projection, FFN, residual connections, and subsequent layers. AegisKV supports runtime KV-headroom control through three core mechanisms: AegisKV-Control, AegisKV-Store, and AegisKV-Flow.

4.2 AegisKV-Control

AegisKV-Control uses serving-side KV pressure and attention-GPU availability as control signals to determine when to activate AO and which requests to route to the AO path.

Offloading trigger. In AegisKV-Control, the scheduler observes the runtime KV pressure of the M-GPUs and the Attn-GPU at each scheduling step, and determines which newly admitted requests should be switched to the AO path. This decision is made based on two pressure signals.

- **M-GPU-side KV pressure.** This signal reflects the KV headroom status on the M-GPU. High KV block usage or frequent preemption indicates that available KV headroom is becoming insufficient. In this regime, keeping requests on the local path can lead to repeated preemption and recomputation overhead.
- **Attn-GPU-side pressure.** When KV block usage on the Attn-GPU increases or scheduling-step latency grows due to offloaded execution, it indicates that the Attn-GPU is becoming a new runtime bottleneck.

Based on the pressure signals, AegisKV dynamically adjusts the scale of the AO path using a target offload degree k. Here, k does not represent the number of newly offloaded requests at the current scheduling step, but rather the target number of AO-path requests that should remain resident on the Attn-GPU.

Let $\mathcal{R}_A$ denote the set of requests currently resident on the Attn-GPU. Then, the number of additional requests that AegisKV can assign to the AO path at the current step is at most $\max(0, k - |\mathcal{R}_A|)$. The target degree k is adjusted within the range $0 \le k \le k_{\max}$, where $k_{\max}$ denotes the upper bound on the number of active requests that the serving engine can sustain simultaneously. When there is no M-GPU pressure, k is set to 0, and new requests are routed to the local path by default.

Algorithm 1 Pressure-Adaptive Request Offloading

Require: newly admitted requests $\mathcal{R}_{\text{new}}$, attention-GPU resident requests $\mathcal{R}_A$, target degree k, max degree $k_{\max}$
Require: M-GPU KV threshold θ_M, attention-GPU KV threshold θ_A, slowdown threshold θ_S, reset window N
Pressure observation
$u_M \leftarrow$ M-GPU KV block usage
$u_A \leftarrow$ attention-GPU KV block usage
$s \leftarrow$ step-time slowdown ratio
$p_M \leftarrow (u_M > \theta_M)$ or preemption observed
$p_A \leftarrow (u_A > \theta_A)$ or $(s > \theta_S)$
Target degree update
if p_A **then**
  $k \leftarrow \max(k - 1, 0)$ ▷ avoid attention-GPU bottleneck
else if p_M **then**
  $k \leftarrow \min(k + 1, k_{\max})$ ▷ relieve M-GPU KV pressure
else if p_M is false for N consecutive steps **then**
  $k \leftarrow 0$ ▷ stop admitting new AO requests
end if
AO-path request selection
$b \leftarrow \max(0, k - |\mathcal{R}_A|)$ ▷ remaining AO resident budget
$\mathcal{R}_{\text{cand}} \leftarrow$ eligible requests in $\mathcal{R}_{\text{new}}$
sort $\mathcal{R}_{\text{cand}}$ by descending estimated KV footprint
mark up to b requests in $\mathcal{R}_{\text{cand}}$ as AO-path requests
return updated k and AO-path assignments

Feedback-based degree control. AegisKV uses a lightweight feedback controller to adjust k incrementally. When M-GPU-side KV pressure is detected and no Attn-GPU-side pressure is present, the scheduler increases k to route more requests to the AO path, restoring serving-side KV headroom and reducing the likelihood of preemption. Conversely, when Attn-GPU-side pressure is detected, the scheduler decreases k to reduce the number of offloaded requests, preventing the AO path from becoming a new decode bottleneck.

If M-GPU-side KV pressure is not observed for N consecutive scheduling steps, the scheduler resets k to 0, and subsequent new requests are routed to the local path. In this case, existing AO-path requests are not immediately migrated back to the local path; instead, they continue executing on the AO path until completion and are naturally drained.

Request selection. When k is updated, AegisKV selects candidates from newly admitted requests to be routed to the AO path. The scheduler prioritizes requests with a large estimated KV footprint, since offloading them allows more M-GPU KV blocks to be released after the prefill stage. The scheduler only assigns the request to the AO path; the actual KV cache transfer is performed after the prefill stage, once the initial KV cache has been created (Step ③ of Figure 2). Algorithm 1 summarizes AegisKV-Control.

4.3 AegisKV-Store

AegisKV-Store manages KV cache placement, ownership, migration, and block release across the M-GPU and the Attn-GPU. When AegisKV-Control selects a request for the AO path, AegisKV-Store

transfers ownership of its KV cache to the Attn-GPU after prefill, after which the request uses the Attn-GPU's KV blocks.

Separate KV cache managers. AegisKV maintains independent KV cache managers on the M-GPU and the Attn-GPU. The M-GPU manager handles KV blocks for both local-path requests and AO-path requests prior to migration. In contrast, the Attn-GPU manager manages KV blocks for AO-path requests after migration is completed.

Each manager performs allocation, extension, and release operations only on its own device-local KV block pool. In addition, each request maintains a KV-location state indicating which GPU its KV cache currently resides on. The scheduler and execution engine use this state to route KV block operations to the appropriate manager. This design separates the KV block namespace between the M-GPU and the Attn-GPU while preserving the original block-based KV management semantics of PagedAttention within each GPU.

KV migration to the Attn-GPU. Requests selected for the AO path first perform prefill on the M-GPU. During prefill, the initial KV cache is stored in blocks allocated by the M-GPU manager. Once prefill is completed, the Attn-GPU manager allocates destination blocks for storing the KV cache of the same request. If destination block allocation fails, the request's location state remains unchanged, and the request stays on the M-GPU, continuing execution on the local path. If allocation succeeds, AegisKV transfers the prefill KV cache from the source blocks on the M-GPU to the destination blocks on the Attn-GPU. Only after the transfer is completed is the request's location state updated to the Attn-GPU, and the M-GPU manager releases the source blocks. This allocate—transfer—update—release sequence preserves the source KV state until migration is successfully completed.

Location-aware memory reclamation. Due to bursty arrivals or unexpectedly long output lengths, KV blocks on a specific GPU may become temporarily insufficient. To reclaim space from the KV pool of the manager experiencing block shortage, AegisKV restricts preemption candidates to requests owned by that same manager. When the M-GPU manager runs out of blocks, preempting AO-path requests whose KV cache resides on the Attn-GPU does not free any M-GPU blocks; therefore, only requests managed by the M-GPU are considered as preemption candidates. Conversely, when the Attn-GPU manager experiences block shortage, only AO-path requests managed by the Attn-GPU manager are considered. This prevents KV pressure on one device from interrupting requests held on the other.

4.4 AegisKV-Flow

AegisKV-Flow is an execution mechanism that performs cross-device transfer and decode attention for AO-path requests at each transformer layer. Rather than separating AO-path requests into an independent batch, AegisKV preserves the existing batching structure, in which each scheduling step may include newly admitted prefill requests together with local-path and AO-path requests already in the decode phase from previous steps.

Layer-wise AO-path execution. Figure 3 illustrates how local-path and AO-path requests are jointly executed at each transformer layer. At each layer, the M-GPU first performs QKV projection for the scheduled batch. Local-path requests perform attention using

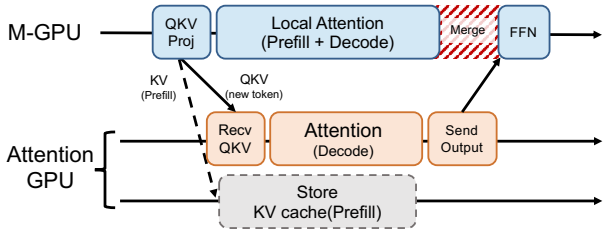


Figure 3: Layer-wise execution under continuous batching. *Execution Flow*: The M-GPU performs QKV projection for the scheduled batch. Prefill KV entries of AO-path requests migrate to the Attn-GPU as future decode state via a background path. For AO-path decode requests, the current query and new KV entry are sent to the Attn-GPU, which computes decode attention over its resident KV cache and returns the attention output to the M-GPU.

the local KV cache on the M-GPU, whereas AO-path decode requests send their current-token queries along with newly generated KV entries to the Attn-GPU. The Attn-GPU appends the new KV entries to its resident KV cache, performs decode attention over this KV cache, and then returns the attention outputs back to the M-GPU.

AegisKV divides cross-GPU data movement into two paths according to their execution dependencies. KV entries generated during prefill are transferred to the Attn-GPU to prepare the resident state for subsequent decode iterations. Since these KV entries are no longer required for generating the current prefill outputs, AegisKV handles them via a background path that can overlap with subsequent layer execution.

In contrast, query and new-KV transfers as well as attention-output returns for AO-path decoding are directly required for the forward progress of the current layer. Therefore, this path cannot be fully handled in the background. AegisKV issues these transfers first and performs local-path attention while the Attn-GPU works, then merges the local-path and AO-path attention outputs into a single batched result. The M-GPU then proceeds with output projection, residual operations, and FFN computation before moving to the next transformer layer.

4.5 Implementation

We implemented AegisKV as a prototype on top of the vLLM v1 engine (v0.10.2). AegisKV implements AegisKV-Control, AegisKV-Store, and AegisKV-Flow in vLLM's Python-level runtime, and does not modify vLLM's existing CUDA/C++ backend or attention kernels.

AegisKV-Control implementation. AegisKV-Control is integrated into each scheduling iteration of the vLLM scheduler and updates the target offload degree k. The KV pressure of the M-GPU and the Attn-GPU is computed as block usage ratios u_M and u_A from the number of free blocks in the block pool managed by each `KVCacheManager`. To track the overhead of offloaded execution, AegisKV maintains exponential moving averages of the wall-clock step time for scheduling steps without offloaded requests and scheduling steps with offloaded requests, respectively. The slowdown signal s is computed as the ratio of these two averages. For request selection, AegisKV uses the number of KV blocks required

by each newly admitted request, as provided by vLLM's KV-cache management metadata, to estimate its KV footprint.

In this prototype, the controller parameters are set to $\theta_M = \theta_A = 0.9$, $\theta_S = 1.5$, and a reset window of $N = 5$ scheduling steps; $k_{\max}$ is set to the same value as the vLLM scheduler's `max_num_seqs`. These thresholds are set to react before a failure mode is fully realized, without reacting so early that offloading is applied where local execution is still preferable: θ_M=θ_A=0.9 leaves room to offload before preemption occurs, and θ_S=1.5 tolerates the moderate step-time increase of a larger decode batch while catching the case where the auxiliary path dominates decode latency.

AegisKV-Store implementation. AegisKV-Store adds a separate `KVCacheManager` instance for the Attn-GPU, so the two GPUs maintain independent KV block pools and block-ID namespaces. Each request maintains a KV-location state, which AegisKV uses to route KV block allocation, update, and release operations to the corresponding `KVCacheManager`. Each manager reuses vLLM's existing block allocation and reclamation logic for its device-local KV block pool.

AegisKV-Flow implementation. AegisKV-Flow assigns the two cross-GPU paths to separate CUDA streams with different priorities. Prefill KV migration is submitted to a low-priority transfer stream as background work, while the query/new-KV transfer, attention-GPU decode, and attention-output return are submitted to a higher-priority decode-side stream. This prioritization reduces the impact of prefill KV migration on query/new-KV transfer and attention-output return during decode. At the merge point, AegisKV uses CUDA events so that output projection and FFN execute only after both the local-path and AO-path attention outputs are ready. In a tensor parallelism environment, each TP-rank process submits the AO-path transfer and decode attention computation for its own shard to the Attn-GPU. As an additional runtime optimization, AegisKV enables NVIDIA Multi-Process Service (MPS) [16] to mitigate the multi-process scheduling overhead on the Attn-GPU.

Attention backend. AegisKV reuses vLLM's existing FlashAttention backend [6] for offloaded attention. It reconstructs the attention-backend invocation metadata for offloaded requests so that the same backend can reference the KV cache resident on the Attn-GPU. Thus, the offloaded path preserves vLLM's support for different attention configurations, including multi-head attention (MHA) [25] and grouped-query attention (GQA) [2].

5 Evaluation

We evaluate AegisKV to understand when and how adaptive attention offloading improves online LLM serving. Our evaluation examines end-to-end performance, the trade-off between KV-headroom expansion and offloading overhead, the effectiveness of its adaptive policy, and the impact of Attn-GPU capability and hardware cost.

5.1 Experimental Setup

We evaluate AegisKV using realistic and synthetic workloads on heterogeneous GPU configurations. Our testbed consists of two NUMA nodes. Unless otherwise specified, we use an NVIDIA L40S as the model GPU (M-GPU) and an NVIDIA RTX 4000 Ada as the attention GPU (Attn-GPU). To study the impact of Attn-GPU capability, we also evaluate configurations that use an NVIDIA L40S as the Attn-GPU. AegisKV supports different attention architectures, including multi-head attention (MHA) and grouped-query attention (GQA). To evaluate this generality, we use two models with different attention structures. Across all experiments, model parameters and KV cache are stored in FP16. We set vLLM's GPU memory utilization, which controls the fraction of GPU memory reserved for the serving engine, to its default value of 0.9 for all systems. This ensures that all systems use the same memory budget while reserving the remaining GPU memory for runtime overheads. Table 2 summarizes the experimental setup.

Table 2: Experimental setup.

GPU Specifications		**Server**	
GPU Model	L40S / RTX 4000 Ada	CPU	2× Xeon Gold 6548Y+ (64C/128T)
GPU Cost	$8,115.65 / $1,543.55	Memory	1 TiB DDR5
FP16 TFLOPS	91.6 / 26.7	NUMA	2 nodes
VRAM	48 GB / 20 GB	PCIe	Gen4 x16 (32 GB/s)
Memory BW	864 GB/s / 360 GB/s		
Software		**Models & Configurations**	
Stack	Ubuntu 24.04, CUDA 12.8, PyTorch 2.8.0	TP1	Llama-2-13B (MHA / 40 KV heads)
Engine	vLLM v0.10.2 (`e08a3a3fd`)	TP4	Qwen2.5-72B (GQA / 8 KV heads)

Model Configuration: Both models use a head dimension of 128. KV heads denote the number of key–value heads.

Workloads. We use ShareGPT and Azure Conversation Trace (Azure) to reflect realistic LLM inference workloads. We filter out samples whose total sequence length exceeds the maximum context length of the evaluated model. ShareGPT provides a mixed input/output length distribution, while Azure is prefill-heavy due to long input sequences. We also include a synthetic workload with a fixed long output length to evaluate decode-heavy scenarios. For all workloads, requests are generated using a Poisson arrival process at a fixed RPS rate to emulate online serving. Detailed characteristics of the workloads used in the evaluation are summarized in Table 3.

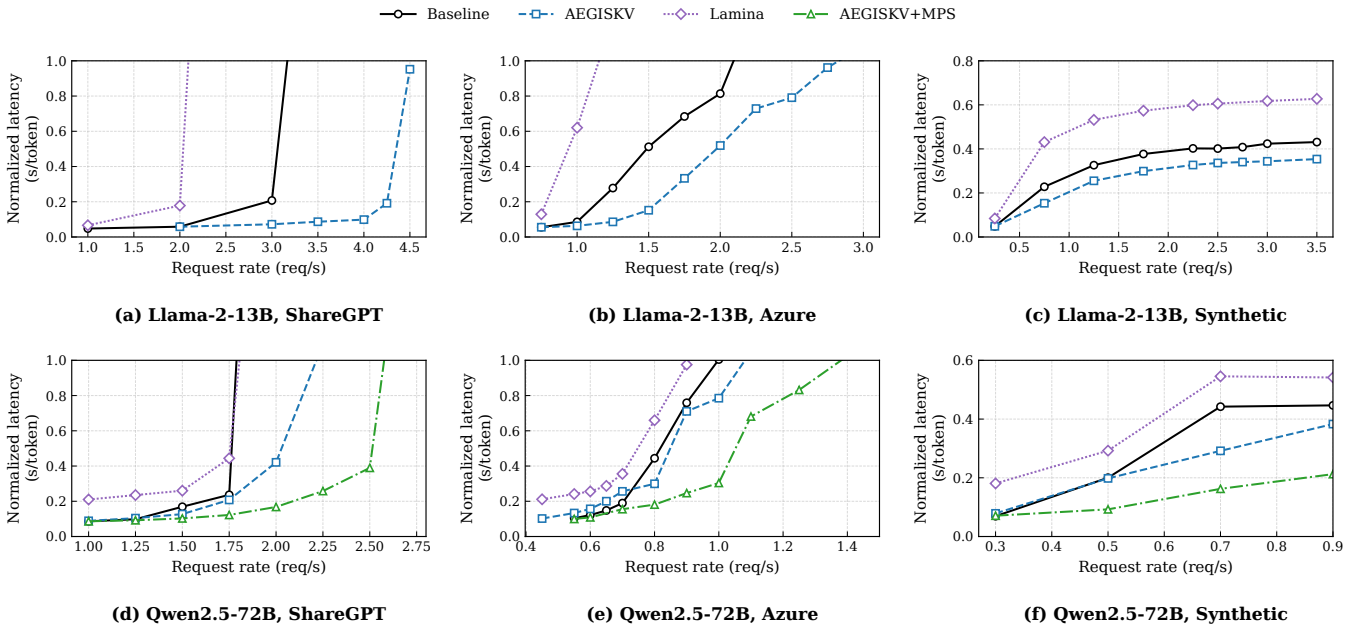
Table 3: Workload characteristics.

Workload	Input	Output	Pattern	Serving Stress
ShareGPT [3]	320.0 / 260.7	195.6 / 192.1	Balanced	Medium KV pressure
Azure [19]	1,154.7 / 753.7	211.1 / 232.5	Prefill-h	High prefill latency
Synthetic	128	1024	Decode-h	Sustained KV growth

For ShareGPT and Azure, each A/B entry reports model-specific average token lengths: A for Qwen2.5-72B and B for Llama-2-13B. In *Workload Patterns*, Prefill-h and Decode-h denote prefill-heavy and decode-heavy workload patterns, respectively.

We compare the following four systems.

- **Baseline [14].** We use vLLM, a representative LLM serving framework, as the baseline.
- **Lamina [4].** A static full attention offloading baseline following Lamina's model-attention disaggregation policy, where every request's decode attention and KV cache reside on the Attn-GPU. Since no vLLM-integrated implementation is public, we implement the policy inside vLLM, sharing all code paths with AegisKV except the offloading policy itself.
- **AegisKV.** AegisKV is our proposed runtime-adaptive system for KV-headroom control. It performs request-level attention offloading based on runtime KV pressure and Attn-GPU capability.



(a) Llama-2-13B, ShareGPT (b) Llama-2-13B, Azure (c) Llama-2-13B, Synthetic

(d) Qwen2.5-72B, ShareGPT (e) Qwen2.5-72B, Azure (f) Qwen2.5-72B, Synthetic

Figure 4: Normalized token latency across request rates and workloads.

Configurations: (a)–(c) evaluate Llama-2-13B under a 1:1 M-GPU-to-Attn-GPU configuration and therefore do not include the MPS setting. (d)–(f) evaluate Qwen2.5-72B under a 4:1 configuration, where four M-GPUs share a single Attn-GPU. *Metric*: Normalized latency is the mean of every request's end-to-end latency divided by its output length, following [28].

- **AegisKV+MPS.** AegisKV+MPS uses NVIDIA MPS to allow a single Attn-GPU to concurrently serve offloaded attention requests from multiple M-GPUs.

5.2 End-to-End Serving Performance

Figure 4 shows the end-to-end serving latency across multiple workloads as we increase the request rate from low-load to high-load regimes. We use these results to analyze how the effectiveness of AO changes with runtime KV pressure.

5.2.1 Under low KV pressure. At lower request rates, where KV pressure remains low, AegisKV achieves latency close to the baseline, while Lamina shows higher latency. This result indicates that local execution on the M-GPU is preferable when the serving GPU has sufficient KV headroom. AegisKV avoids unnecessary offloading in this regime, whereas Lamina's static full AO continuously routes attention to the Attn-GPU regardless of runtime pressure, adding cross-GPU transfer and auxiliary execution overheads.

5.2.2 Under high KV pressure. At higher request rates, where KV pressure becomes high, the latency behavior depends on workload characteristics.

Balanced and prefill-heavy workloads. Figure 4a and Figure 4b show the results for Llama-2-13B. Both Baseline and Lamina exhibit saturation behavior, where latency increases sharply beyond a certain request rate. In contrast, AegisKV delays the onset of saturation by up to 1.42× over Baseline and 2.13× over Lamina, sustaining stable latency at higher request rates. This is because AegisKV selectively offloads requests only under KV pressure, preserving M-GPU KV headroom while avoiding unnecessary auxiliary-path overhead. We observe a similar trend for Qwen2.5-72B. As shown in Figure 4d and Figure 4e, AegisKV delays the saturation point by up to 1.14× compared to Lamina. Unlike Static Full AO, which routes all decode attention to the Attn-GPU, AegisKV adapts the offloading decision to runtime pressure and effectively combines local and AO paths to sustain a larger active batch.

Decode-heavy workload. Unlike the previous workloads, the decode-heavy workload shows a more gradual increase in latency. This is because long output generation makes decode the dominant execution phase, and the system throughput gradually saturates as the decode batch size stabilizes. In Figure 4c, at 3.0 RPS, AegisKV achieves a normalized latency of 0.344 s/token, improving over Baseline (0.424 s/token) and Lamina (0.618 s/token) by 1.23× and 1.80×, respectively. This result shows that AegisKV can relieve KV memory pressure and maintain a larger decode batch even under long-output generation. We observe the same trend for the Qwen model in Figure 4f, indicating that AegisKV's memory-adaptive offloading strategy remains effective for decode-heavy workloads.

Effect of MPS. AegisKV+MPS further improves performance over AegisKV across all workloads. In particular, for the prefill-heavy workload, AegisKV+MPS delays the onset of latency saturation by up to 1.43× compared to the baseline, and sustains the same latency at up to 1.25× higher RPS than AegisKV. This improvement results from better Attn-GPU utilization when multiple M-GPUs share a single Attn-GPU via NVIDIA MPS. We provide a detailed analysis in §5.3.2.

5.3 KV Headroom vs. Offloading Overhead

Attention offloading trades increased KV headroom for additional decode overhead. We analyze how AegisKV balances this trade-off through adaptive offloading.

5.3.1 ***KV-Headroom Expansion Benefit.*** Figure 5a and Figure 5c show the TTFT trend as the offered load increases. AegisKV achieves 5.94× lower TTFT than Baseline, and AegisKV+MPS further achieves 15.07× lower TTFT (Qwen Synthetic, RPS=0.7). This is because selective offloading relieves KV pressure on the M-GPU, delaying KV-cache saturation and enabling earlier prefill admission for waiting requests while sustaining higher request concurrency.

Lamina also expands KV capacity by offloading every request's KV cache and decode attention to the Attn-GPU. However, once the Attn-GPU becomes saturated, Lamina cannot exploit remaining KV headroom on the M-GPU. This leads to lower throughput and higher latency than AegisKV under high load.

These results indicate that AegisKV's benefit is not simply from distributing attention computation. Rather, AegisKV improves performance by adaptively managing runtime KV headroom to maintain higher request concurrency.

5.3.2 ***Attention Offloading Overhead.*** Figure 5b and Figure 5d show how TPOT changes as the request rate increases. Attention offloading introduces additional data movement in the decode path, including current-token query/key/value transfer and attention-output transfer, thereby increasing communication overhead. Moreover, as offloading allows the system to keep more decode requests active, the per-iteration attention compute load increases, and newly admitted prefill requests can further interfere with decode latency [1].

This overhead is particularly pronounced in Lamina. In our results, Lamina increases TPOT by up to 3.23× over Baseline (Qwen ShareGPT, RPS=2.0), and its latency begins to increase sharply from RPS=1.75. This is because excessive offloading makes the decode-latency overhead outweigh the throughput benefit gained from a larger active batch.

This effect becomes more significant under the TP=4 configuration. In this setting, a single Attn-GPU must serve the offloaded attention compute load from four TP shards, making the Attn-GPU an attention-compute bottleneck. Such a bottleneck is more likely when the sharing ratio between M-GPUs and the Attn-GPU is high. As a result, attention kernels on the Attn-GPU become more serialized, which increases overall decode latency.

In contrast, AegisKV dynamically controls the degree of offloading by considering both KV pressure and Attn-GPU compute load. AegisKV performs attention offloading only when the benefit from additional KV headroom outweighs the added decode overhead. Under low load, most requests remain on the M-GPU, allowing AegisKV to maintain TPOT close to Baseline. Under high load, AegisKV selectively activates offloading and mitigates TPOT inflation compared to Lamina.

AegisKV further uses NVIDIA MPS to reduce the Attn-GPU bottleneck. MPS allows CUDA workloads from multiple processes to execute more concurrently, improving Attn-GPU utilization and reducing kernel serialization. As a result, AegisKV+MPS achieves lower TTFT and TPOT than AegisKV alone across all workloads.

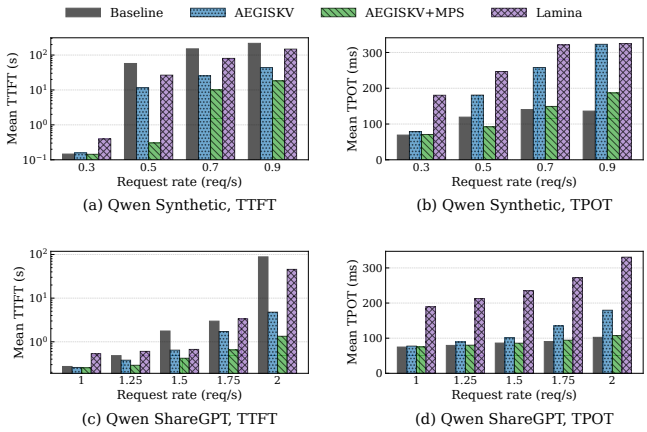


Figure 5: Latency comparison of Qwen serving configurations across request rates.

5.4 Offloading Policy Analysis

To evaluate the effectiveness of AegisKV's selective offloading policy, we compare it against two alternative policies as follows:

- **Static-Long** offloads a fixed fraction of requests. We set the offloading ratio to 30% based on the relative memory bandwidth between the M-GPU and the Attn-GPU. As in AegisKV, Static-Long prioritizes requests with larger KV-cache footprints as offloading candidates.
- **Static-FCFS** uses the same fixed offloading ratio as Static-Long, but does not select offloading candidates based on their KV-cache footprint. Instead, it offloads requests in first-come-first-served (FCFS) order, following vLLM's default scheduling order.

As shown in Figure 6, as workload intensity increases, the latency of the two static policies grows much more rapidly than that of AegisKV. The comparison between AegisKV and Static-Long demonstrates that dynamically controlling the offloading degree is critical. A fixed offloading ratio cannot adapt to varying KV cache demands, failing to secure sufficient KV headroom under high KV pressure. By dynamically adjusting the offloading degree based on runtime KV pressure, AegisKV enables effective KV headroom management.

The comparison between Static-FCFS and Static-Long further shows that the choice of offloading candidates is also important. Since FCFS-based offloading ignores input length, it provides limited KV-headroom relief. Prioritizing requests with longer input sequences can reclaim more KV blocks from the M-GPU, enabling the serving runtime to sustain higher request concurrency.

5.5 Impact of Attention-GPU Capability

To analyze how Attn-GPU capability affects offloading performance, we replace the RTX 4000 Ada with a more capable L40S as the Attn-GPU and run experiments using Qwen2.5-72B with the Azure workload. Figure 7 shows that the stronger Attn-GPU lowers normalized latency at the same request rate and shifts the latency curve to the right, enabling higher sustainable request rates under the same latency target. Under the AegisKV+MPS configuration, replacing the RTX 4000 Ada with an L40S Attn-GPU increases the

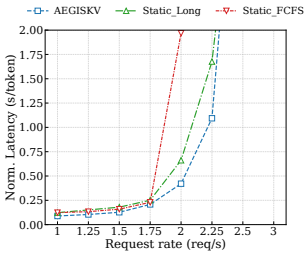


Figure 6: Latency across offloading policies (Qwen2.5-72B, ShareGPT).

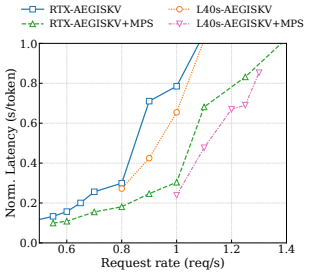


Figure 7: Latency across Attn-GPU configurations (Qwen2.5-72B, Azure).

sustainable request rate from 1.0 to 1.2 RPS around a normalized latency of 0.65 s/token. This is because L40S provides larger KV-cache headroom and reduces offloaded attention execution time through higher compute capability.

5.6 Cost-Efficiency Analysis

We analyze how effectively AegisKV improves serving throughput relative to the additional hardware cost. Both metrics below are based on the *saturation-point RPS*, the highest request rate the system sustains before entering the latency saturation region.

Metric 1: Overall Cost Efficiency. Metric 1 measures the absolute cost efficiency of the overall system in terms of request throughput per unit cost, i.e., RPS per dollar.

$$\text{Metric 1} = \frac{\text{RPS}_{\text{proposed}}/\text{Cost}_{\text{proposed}}}{\text{RPS}_{\text{base}}/\text{Cost}_{\text{base}}} \tag{1}$$

A Metric 1 value greater than 1.0× indicates that the proposed system provides higher throughput for the same cost than the baseline.

Metric 2: Marginal ROI. Metric 2 measures the marginal return on investment of adding the Attn-GPU.

$$\text{Metric 2} = \frac{\Delta\text{RPS}/\text{RPS}_{\text{base}}}{\Delta\text{Cost}/\text{Cost}_{\text{base}}} \tag{2}$$

Here, $\Delta\text{RPS} = \text{RPS}_{\text{proposed}} - \text{RPS}_{\text{base}}$, and ΔCost denotes the additional cost of the RTX 4000 Ada Attn-GPU. We define $\text{Cost}_{\text{proposed}} = \text{Cost}_{\text{base}} + \Delta\text{Cost}$.

Table 4: Performance and cost-efficiency summary.

Model	Base RPS	Proposed RPS	Metric 1	Metric 2
Llama	3.00 (1×L40S)	4.25 (+1×RTX4000)	1.19×	2.19
Qwen	1.75 (4×L40S)	2.50 (+1×RTX4000)	1.36×	9.01

Table 4 reports both metrics on the ShareGPT workload for Llama and Qwen, using representative online retail GPU prices at the time of evaluation as the cost basis (listed in Table 2).

Metric 1 shows that AegisKV achieves higher cost efficiency than Baseline across both model configurations. In particular, AegisKV improves throughput per unit cost by up to 19% for Llama and 36% for Qwen.

In the Qwen configuration, adding one RTX 4000 Ada GPU increases the total system cost by only 4.8%, but improves the saturation-point RPS by 42.9%. As a result, Metric 2 reaches 9.01, indicating that AegisKV can expand the serving capacity of an existing GPU cluster with a high marginal return on investment.

6 Related Work

Scheduling for LLM serving. Prior LLM serving systems improve online inference efficiency through request scheduling and continuous batching. Orca [28] introduces iteration-level scheduling, vLLM [14] improves KV cache management with PagedAttention, and Sarathi-Serve [1] co-schedules chunked prefill and decode requests. Recent systems further incorporate SLO-aware and request-length-aware scheduling to improve serving goodput and reduce latency variance [20, 21, 24, 29]. However, these approaches primarily optimize request admission and batching within a fixed GPU memory budget, without addressing runtime KV cache pressure itself. In contrast, AegisKV selectively offloads KV cache and attention computation to an auxiliary GPU under KV pressure, extending the effective execution path to provide additional runtime KV headroom while preserving the existing continuous-batching framework.

Attention offloading. CPU-based AO systems such as FastDecode [9], NEO [10], and HGCA [7] offload KV cache and attention computation to host CPUs, but their performance is limited by CPU compute capability and memory bandwidth under high concurrency [18]. Other approaches use specialized hardware such as CSDs, CXL-attached PNM, or PIM memory to accelerate attention offloading [13, 17, 18], but these require hardware support that is difficult to deploy in existing GPU-centric clusters. Recent GPU-based AO systems instead repurpose GPUs as attention devices. Weaver [8] offloads attention at a fixed ratio and depends on the availability of residual GPU memory from cold-model instances. Lamina [4] instead uses dedicated memory-optimized GPU pools with a static full-offload policy that can introduce unnecessary overhead under low KV pressure. In contrast, AegisKV dynamically determines when to activate the AO path and which requests to offload based on runtime KV pressure, without requiring specialized hardware or auxiliary model execution.

7 Conclusion

This paper presented AegisKV, an adaptive attention offloading system for online LLM serving under KV-cache pressure. By controlling request-level offloading based on KV headroom, AegisKV reduces preemption and recomputation overhead while preserving serving-side KV headroom under high request concurrency. Our results show that adaptive GPU-based attention offloading enables incremental serving-capacity expansion by turning heterogeneous auxiliary GPUs into runtime KV-headroom resources.

Acknowledgments

This work was supported by the Korea government (MSIT) through the National Research Foundation of Korea (NRF) (RS-2024-00416666 and RS-2025-00564249) and the Institute of Information & Communications Technology Planning & Evaluation (IITP) (IITP-RS-2026-25547954).

GenAI Disclosure Statement

Generative AI tools were used only for language editing, including grammar correction and improving the clarity of the writing. All content was reviewed and verified by the authors.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation* (Santa Clara, CA) *(OSDI '24)*. USENIX Association, Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[2] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing* (Singapore). Association for Computational Linguistics, Singapore, 4895–4901. doi:10.18653/v1/2023.emnlp-main.298

[3] anon8231489123. 2023. ShareGPT_Vicuna_unfiltered. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered.

[4] Shaoyuan Chen, Wencong Xiao, Yutong Lin, Mingxing Zhang, Yingdi Shan, Jinlei Jiang, Kang Chen, and Yongwei Wu. 2025. *Efficient Heterogeneous Large Language Model Decoding with Model-Attention Disaggregation.* arXiv:2405.01814 doi:10.48550/arXiv.2405.01814

[5] Sumit Kumar Dam, Choong Seon Hong, Yu Qiao, and Chaoning Zhang. 2024. *A Complete Survey on LLM-based AI Chatbots.* arXiv:2406.16937 doi:10.48550/arXiv.2406.16937

[6] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[7] Weishu Deng, Yujie Yang, Peiran Du, Lingfeng Xiang, Zhen Lin, Chen Zhong, Song Jiang, Hui Lu, and Jia Rao. 2025. *HGCA: Hybrid GPU-CPU Attention for Long Context LLM Inference.* arXiv:2507.03153 doi:10.48550/arXiv.2507.03153

[8] Shiwei Gao, Qing Wang, Shaoxun Zeng, Youyou Lu, and Jiwu Shu. 2025. Weaver: Efficient Multi-LLM Serving with Attention Offloading. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, Boston, MA, 587–595. https://www.usenix.org/conference/atc25/presentation/gao

[9] Jiaao He and Jidong Zhai. 2024. *FastDecode: High-Throughput GPU-Efficient LLM Serving using Heterogeneous Pipelines.* arXiv:2403.11421 doi:10.48550/arXiv.2403.11421

[10] Xuanlin Jiang, Yang Zhou, Shiyi Cao, Ion Stoica, and Minlan Yu. 2025. NEO: Saving GPU Memory Crisis with CPU Offloading for Online LLM Inference. In *Proceedings of Machine Learning and Systems*, M. Zaharia, G. Joshi, and Y. Lin (Eds.), Vol. 7. MLSys. https://proceedings.mlsys.org/paper_files/paper/2025/file/66a026c0d17040889b50f0dfa650e5e0-Paper-Conference.pdf

[11] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. *S^3: Increasing GPU Utilization during Generative Inference for Higher Throughput.* arXiv:2306.06000 doi:10.48550/arXiv.2306.06000

[12] Aditya K. Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. 2025. POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 897–912. doi:10.1145/3676641.3715996

[13] Dowon Kim, MinJae Lee, Janghyeon Kim, HyuckSung Kwon, Hyeonggyu Jeong, Sang-Soo Park, Minyong Yoon, Si-Dong Roh, Yongsuk Kwon, Jinin So, and Jungwook Choi. 2025. Scalable Processing-Near-Memory for 1M-Token LLM Inference: CXL-Enabled KV-Cache Management Beyond GPU Limits. In *2025 34th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Irvine, CA, USA). IEEE Press, 1–13. doi:10.1109/PACT65351.2025.00013

[14] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[15] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 97, 13 pages. doi:10.1145/3597503.3639187

[16] NVIDIA Corporation. 2026. CUDA Multi-Process Service. Accessed: May 2026. https://docs.nvidia.com/deploy/mps/latest/index.html#organization-of-this-document

[17] Xiurui Pan, Endian Li, Qiao Li, Shengwen Liang, Yizhou Shan, Ke Zhou, Yingwei Luo, Xiaolin Wang, and Jie Zhang. 2025. InstAttention: In-Storage Attention Offloading for Cost-Effective Long-Context LLM Inference. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1510–1525. doi:10.1109/HPCA61900.2025.00113

[18] Jaehyun Park, Jaewan Choi, Kwanhee Kyung, Michael Jaemin Kim, Yongsuk Kwon, Nam Sung Kim, and Jung Ho Ahn. 2024. AttAcc! Unleashing the Power of PIM for Batched Transformer-Based Generative Model Inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 103–119. doi:10.1145/3620665.3640422

[19] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture* (Buenos Aires, Argentina) *(ISCA '24)*. IEEE Press, 118–132. doi:10.1109/ISCA59077.2024.00019

[20] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Chandra Narayanaswami, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2024. Queue Management for SLO-Oriented Large Language Model Serving. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 18–35. doi:10.1145/3698038.3698523

[21] Haiying Shen and Tanmoy Sen. 2025. *AccelGen: Heterogeneous SLO-Guaranteed High-Throughput LLM Inference Serving for Diverse Applications.* arXiv:2503.13737 doi:10.48550/arXiv.2503.13737

[22] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism.* arXiv:1909.08053 doi:10.48550/arXiv.1909.08053

[23] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 173–191. https://www.usenix.org/conference/osdi24/presentation/sun-biao

[24] Yinghao Tang, Tingfeng Lan, Xiuqi Huang, Hui Lu, and Wei Chen. 2025. *SCORPIO: Serving the Right Requests at the Right Time for Heterogeneous SLOs in LLM Inference.* arXiv:2505.23022 doi:10.48550/arXiv.2505.23022

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. *Attention Is All You Need.* arXiv:1706.03762 doi:10.48550/arXiv.1706.03762

[26] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 640–654. doi:10.1145/3694715.3695948

[27] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. *ReAct: Synergizing Reasoning and Acting in Language Models.* arXiv:2210.03629 doi:10.48550/arXiv.2210.03629

[28] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA) *(OSDI '22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[29] Wei Zhang, Zhiyu Wu, Yi Mu, Rui Ning, Banruo Liu, Nikhil Sarda, Myungjin Lee, and Fan Lai. 2025. *JITServe: SLO-aware LLM Serving with Imprecise Request Information.* arXiv:2504.20068 doi:10.48550/arXiv.2504.20068

[30] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NeurIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 2000, 27 pages.