RHEOSERVE: CPU-GPU COLLABORATIVE SPARSE ATTENTION FOR EFFICIENT LLM DECODING

Anonymous authors
Paper under double-blind review

ABSTRACT

Sparse attention is a promising approach for efficient LLM decoding over large KV caches, yet managing these caches remains a bottleneck: eviction-based methods sacrifice accuracy, while retrieval-based offloading is constrained by limited PCIe bandwidth, thereby bottlenecking decoding throughput. To address this, we present RheoServe, a CPU–GPU collaborative sparse-attention framework that actively leverages host memory bandwidth via block-level cache management. RheoServe adopts a compute offload paradigm: rather than moving massive data to the GPU, it moves compute to the data, executing sparse attention directly on DDR-resident blocks and merging results via Log-Sum-Exp (LSE) reduction. Furthermore, RheoServe employs a unified cache mechanism that intelligently manages block placement, improving cache hit rate to compensate for the bandwidth gap. Extensive evaluations demonstrate that RheoServe maintains reasoning accuracy and achieves $1.40\times$ and $1.57\times$ the peak throughput of matched PCIe Retrieval for Qwen3-4B and Qwen3-8B, respectively.

1 INTRODUCTION

Large language models are more powerful than ever. But training bigger models alone is no longer enough. Test-time scaling DeepSeek-AI (2025) offers a new path forward: instead of scaling model size, we let models think longer. By generating extended reasoning chains, they can solve problems that were previously out of reach, and difficult problems can require long sequences of autoregressive reasoning steps.

This growth in sequence length imposes a severe bottleneck on serving infrastructure, as the Key-Value (KV) cache grows linearly with context length and must be accessed at every decoding step. For instance, a single 32K-token request with Qwen3-8B (Team, 2025) using Grouped-Query Attention (Ainslie et al., 2023) requires 4.5 GB of KV-cache memory, and this pressure is significantly amplified during batched decoding, where serving just 8–16 concurrent sequences can rapidly exhaust available GPU memory.

The question is unavoidable: where do we store and compute over KV that no longer fits on GPU? Host-memory offload is a natural answer for storing cold (seldom visited) KV in CPU DDR. However, current systems have not fully exploited this heterogeneous hierarchy. Our analysis identifies three key gaps in existing designs that RheoServe aims to bridge (Figure 1).

Opportunity 1: The SW-HW Co-design Gap. Current sparse attention approaches optimize for orthogonal goals. Eviction-based methods like StreamingLLM Xiao et al. (2024), H2O Zhang et al. (2023) and SnapKV Li et al. (2024) operate under a fixed GPU budget, ensuring efficiency but potentially limiting reasoning by discarding context. Heterogeneous systems instead distribute KV blocks across CPU and GPU memory: MagicPIG Chen et al. (2025) and HGCA Deng et al. (2025) use collaborative CPU–GPU attention computation, whereas ArkVale Chen et al. (2024) and SparseServe Zhou et al. (2026) retrieve selected host-resident KV blocks to the GPU over PCIe. Both families, however, face a common challenge: GPU and host memory differ substantially in bandwidth, and attention patterns shift dynamically across decoding steps. This raises a key question: how can we design a block-level KV management mechanism that adapts to these changing patterns and bandwidth asymmetries, keeping hot blocks in high-bandwidth GPU memory while efficiently serving cold blocks from the host?

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

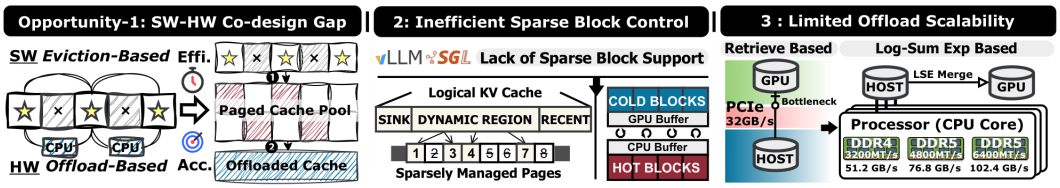


Figure 1: Opportunities in efficient LLM serving.

Opportunity 2: Lack of Native Support for Sparse Block Control. vLLM's PagedAttention Kwon et al. (2023) manages KV blocks within GPU memory, but does not jointly schedule sparse block selection and migration across GPU and host memory. This gap makes it difficult to implement dynamic sparsity efficiently, especially when important context blocks are scattered throughout the sequence and may need to move between GPU and host memory. This raises a natural question: can we introduce an abstraction layer that decouples logical context from physical placement, automatically managing the migration of hot and cold blocks to optimize data locality?

Opportunity 3: Bandwidth Misalignment in Offload Systems. When GPU memory is exhausted, existing systems relying on retrieval move selected KV blocks across the PCIe bus back to the GPU for computation. This creates a bottleneck at the interconnect: even though host DDR bandwidth often significantly exceeds PCIe bandwidth ($\sim$51.2 GB/s vs $\sim$32 GB/s), the effective throughput is capped by the narrower PCIe link, leaving abundant host memory bandwidth and CPU compute cycles idle. This leads to a question: can we exploit the higher host bandwidth by computing attention in place?

RheoServe Framework. We address these challenges by introducing *compute offload*: rather than retrieving cold KV blocks back to the GPU, RheoServe processes them directly on the CPU and merges partial attention results via Log-Sum-Exp reduction. This paradigm removes PCIe retrieval from the decoding critical path, replacing it with parallel heterogeneous computation.

Our contributions are as follows:

Contribution 1: We propose a Logical KV Cache abstraction that decouples physical memory management from inference logic. By automating the dynamic migration of hot and cold blocks, this abstraction maintains high GPU hit rates for sparse attention.

Contribution 2: We implement a compute offload mechanism that leverages underutilized host resources (CPU cores and DDR bandwidth) for attention computation. This design bypasses retrieval bottlenecks when the KV cache exceeds GPU memory.

Our end-to-end evaluation shows that our compute-offload design provides a clear throughput advantage over PCIe retrieval. On an RTX 3090, RheoServe delivers $1.40\times$ the peak throughput of PCIe Retrieval. It also reaches $1.64\times$ and $1.44\times$ the peak throughput of offload-enabled vLLM and SGLang on the same GPU. These results demonstrate that processing KV blocks where they reside, rather than retrieving them across PCIe, makes host memory an effective execution tier for sparse-attention serving.

2 Background and Related Work

Attention Merge via LSE

Compute offload requires the ability to merge partial attention results computed independently over disjoint token sets. Given query q and keys/values partitioned into $K = K_1 \cup K_2$ (disjoint), standard attention computes:

$$o = \frac{\sum_i \exp(q^\top k_i) v_i}{\sum_i \exp(q^\top k_i)} \tag{1}$$

Computing this directly requires access to all keys and values. Instead, we compute each partition's statistics (m_k, ℓ_k, o_k), where $m_k = \max_{i \in K_k} q^\top k_i$ is the max score, $\ell_k = \sum_{i \in K_k} \exp(q^\top k_i - m_k)$ is the normalized sum, and o_k is the partial weighted output. The exact global result is then:

$$m = \max(m_1, m_2), \quad \ell = \ell_1 e^{m_1 - m} + \ell_2 e^{m_2 - m} \tag{2}$$

$$o = \frac{\ell_1 e^{m_1 - m} o_1 + \ell_2 e^{m_2 - m} o_2}{\ell} \tag{3}$$

This allows GPU and CPU to independently compute attention over their respective token subsets and merge results.

Sparse Attention Algorithms

The softmax attention distribution is often sparse in practice: a small subset of tokens contributes most of the attention mass. Leveraging this sparsity has been a central theme in recent LLM serving research. Early eviction-based methods such as StreamingLLM Xiao et al. (2024), H2O Zhang et al. (2023), and SnapKV Li et al. (2024) manage a fixed GPU cache budget by permanently dropping less important tokens, achieving strong memory efficiency. However, this approach risks discarding information needed for future reasoning steps, leading to accuracy degradation on long-range dependencies. To address this, dynamic sparse-attention methods like Quest Tang et al. (2024), MagicPIG Chen et al. (2025), HashAttention Desai et al. (2025), and SeerAttention-R Gao et al. (2025) evaluate token importance at each decoding step, retaining relevant context without permanently evicting it.

Among these, we adopt SeerAttention-R Gao et al. (2025), which uses a self-distilled gating network to produce compact metadata, leaving more GPU capacity for KV blocks. Crucially, this choice instantiates rather than defines RheoServe: the selector can be replaced while retaining our heterogeneous KV-management and execution substrate. The growing adoption of native sparse attention in models like DeepSeek-V4 DeepSeek-AI (2026) and GLM-5 Z.ai (2026) confirms that sparsity is evolving from an inference-time add-on to a built-in model capability, reinforcing the direction of RheoServe.

Efficient LLM Serving

Efficient LLM decoding over growing KV caches requires addressing both memory management and computational bottlenecks. vLLM Kwon et al. (2023) introduced PagedAttention to manage fragmented KV memory, while FlashInfer Ye et al. (2025) provides a composable attention engine with native support for LSE-based result merging. Recent works like MagicPIG Chen et al. (2025) and HGCA Deng et al. (2025) have explored CPU–GPU collaboration via LSE merge, yet they lack the cache management needed for high-throughput inference.

To scale beyond GPU memory, systems such as ArkVale Chen et al. (2024) and SparseServe Zhou et al. (2026) adopt a retrieval-based paradigm: offloading cold blocks to host memory and fetching them to the GPU on demand. While effective for storage, these systems treat host memory as passive capacity, leaving decode throughput bounded by PCIe transfer latency. In contrast, RheoServe adopts a compute-offload strategy that executes attention directly on the CPU over DDR-resident data, leveraging heterogeneous compute resources to bypass this bottleneck.

3 RheoServe System Design

System Overview

RheoServe comprises three tightly integrated modules (Figure 2): a Dual-Track Cache that unifies logical addressing with sparse block selection, a Block Controller that manages cross-tier migration, and a Runtime Scheduler that manages transaction lifecycles through dynamic cache allocation and deallocation. The first two modules together replace the standard attention kernel with a cache-aware execution pipeline, while the Runtime Scheduler coordinates them at the batch level.

Dual-Track Cache. The Dual-Track Cache (Figure 2(a)) serves as the system's unified storage layer, managing massive KV data distributed across GPU and CPU at block granularity (e.g., 32 tokens). It abstracts this complexity via a Logical KV Cache, while hosting a lightweight Key Index on the GPU to accelerate sparse block selection.

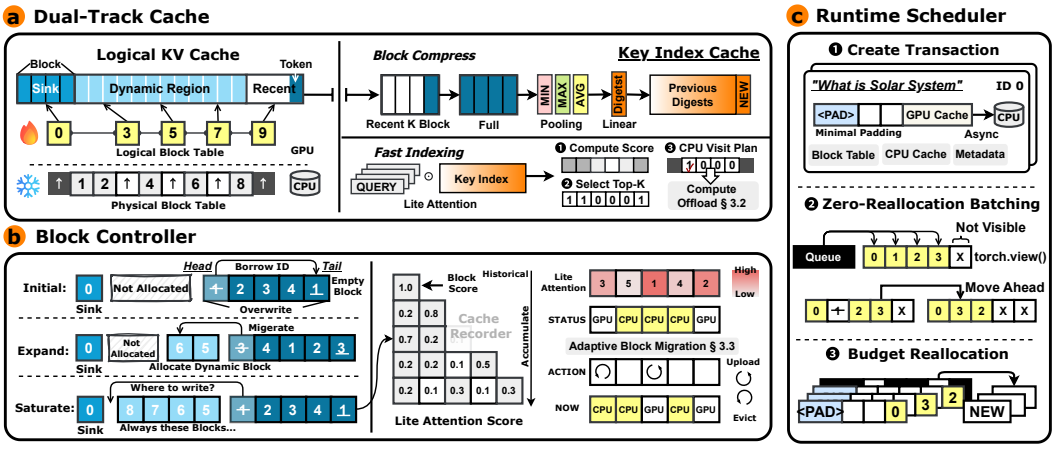


Figure 2: RheoServe System Overview. (a) Dual-Track Cache: Provides a Logical KV Cache abstraction and accelerates scoring via the Key Index. (b) Block Controller: Orchestrates allocation and cross-tier migration. (c) Runtime Controller: Manages lifecycles and dynamic cache scheduling.

Structurally, the logical cache consists of three functional regions (Figure 2(a), left): an Attention Sink Xiao et al. (2024) for stability, a Recent Window for local context, and a Dynamic Region for long-range dependencies. We designate the combination of the sink and the recent window as the Reserved Region, which occupies a fixed portion of GPU memory.

The extensive intermediate context, however, necessitates virtualization to bridge the gap between massive KV requirements and limited GPU physical memory. Accordingly, we design the Dynamic Region as a virtualized address space that maps to hot blocks on demand, while the complete history is backed by large-capacity CPU memory. By unifying these three regions, we establish a logical block table that provides a continuous view to the attention kernel, shielding it from the physical complexity managed by the Physical Block Table.

Figure 2(a) (right) illustrates the Key Index construction. Whenever the recent token buffer fills a block (e.g., 32 tokens), it is compressed to generate a Key Digest. To ensure the digest accurately reflects block content, we adopt the method from SeerAttention-R Gao et al. (2025): we first compute min/max/mean pooling statistics to bound the token distribution, then apply a learned lightweight projection to fuse these statistics into the final Key Digest. This compact digest is appended to the on-device Key Index, consuming only 1/32 of the original Key Cache size.

Leveraging the Key Index, we can perform fast indexing at decode time. Incoming query tokens perform lightweight attention against this index, significantly reducing the memory footprint compared to full KV Cache while yielding block-wise relevance scores. We then perform top-K selection to identify the most relevant blocks. With the GPU handling hot blocks via the Logical KV Cache, we generate a CPU visit plan specifically for the missed blocks, as detailed in the Compute offload subsection.

Block Controller. The Block Controller manages the physical placement of blocks across GPU and CPU memory, preserving the semantics of the Logical KV Cache through a lifecycle of three distinct phases (Figure 2(b), left):

❶ *Reserved Phase.* Initially, the system operates within the Reserved Region. During this phase, the recent block buffer functions as a circular queue where the oldest block ID is recycled to the tail for new tokens, overwriting GPU data. However, unlike pure eviction methods, a complete backup copy is persisted on the CPU to ensure overwritten blocks remain available for compute offload.

❷ *Expansion Phase.* Once dynamic capacity is allocated, the system enters an expansion phase. Instead of overwriting the head block in the recent buffer, the controller migrates it to a newly allocated block in the Dynamic Region, preserving mid-sequence context on the GPU.

❸ *Saturation Phase.* Finally, when the Dynamic Region is exhausted, rather than discarding critical long-range dependencies, we implement an automatic block migration strategy (Figure 2(b), right).

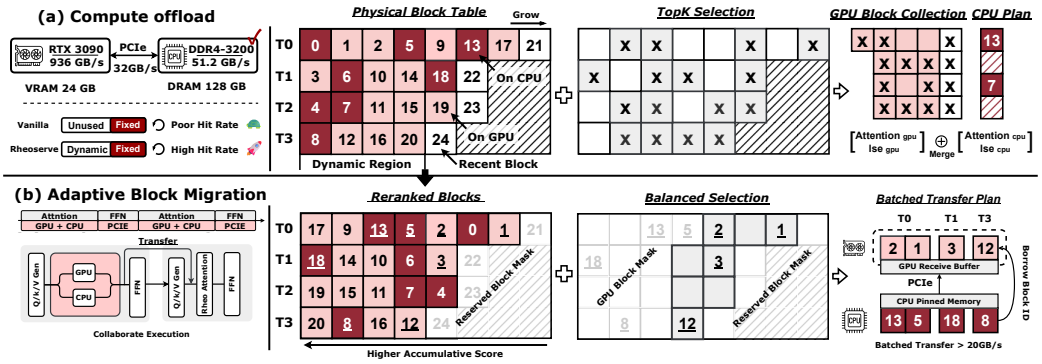


Figure 3: Detailed System Mechanics. (a) Compute offload orchestrates collaborative execution. (b) Adaptive Block Migration optimizes block placement through score-based ranking, selection, and asynchronous migration.

Historical attention scores guide promotion of high-scoring blocks to the GPU and eviction of lower-scoring blocks to the CPU. The current implementation smooths these scores exponentially. The smoothing rule is a replaceable policy choice within the migration mechanism. This strategy is detailed in the Adaptive Block Migration subsection.

Runtime Scheduler. RheoServe treats sequences as ❶ transactions, comprising prompt content, KV cache references, and metadata (e.g., Key Index, Block Table). To maximize decoding throughput, we adopt a batched execution strategy. Standard padding strategies (e.g., Hugging Face Transformers) force uniform sequence lengths, which can corrupt attention sink functionality by inadvertently filling sinks with padding tokens. To address this, RheoServe implements a minimal padding strategy with independent rotary embeddings Su et al. (2024), allowing sequences of varying lengths to execute in parallel without interference. ❷ For runtime stability, we implement zero-reallocation batching by preallocating a fixed-size metadata buffer. Transactions are registered into fixed slots, and `torch.view()` restricts computation to active regions. Upon transaction completion (e.g., <EOS>), we employ a swap-and-pop removal strategy: the last active transaction is moved to fill the vacated slot. This avoids the high memory cost of slicing and merging metadata tensors, which can otherwise lead to memory explosion. Finally, ❸ we immediately reclaim freed blocks from the Dynamic Region and redistribute them to expand the budget of remaining active transactions based on their context length.

COMPUTE OFFLOAD

As sequences grow, some blocks inevitably reside on the host. This subsection details how RheoServe orchestrates attention across both tiers through compute offload (Figure 3(a)). The choice of compute offload over retrieval is driven by bandwidth economics: a typical dual-channel DDR4-3200 configuration delivers 51.2 GB/s, whereas PCIe 4.0 offers 32 GB/s. Retrieval-based systems must transfer selected KV blocks back to the GPU through this slower path; RheoServe instead executes attention where the data resides. To further reduce the workload on the CPU path, RheoServe maintains a Dynamic Region that caches frequently accessed blocks on the GPU, minimizing the number of blocks requiring CPU-side computation.

Figure 3(a) (right) illustrates the execution flow. The Physical Block Table tracks the location of all blocks across running transactions (e.g., T1–T4). Given an incoming query, RheoServe performs lightweight attention against the Key Index to obtain per-block relevance scores, then selects the top-K blocks as a sparse attention mask. Applying this mask to the Block Table quickly identifies which cold blocks require CPU-side computation. GPU and CPU then compute attention independently over their respective block subsets, merging results exactly via Log-Sum-Exp reduction as derived in Section 2.

Adaptive Block Migration

Because RheoServe executes attention where the data resides, block placement directly determines which tier handles each computation. To keep this placement up to date without stalling inference, we introduce a Collaborative Execution strategy. As illustrated in Figure 3(b) (left), RheoServe orchestrates three parallel execution paths: host-side attention work is launched alongside a forked GPU compute stream, while a dedicated transfer path migrates blocks between attention operations. This parallelism uses host compute and PCIe bandwidth concurrently and hides much of the migration latency behind the core inference path.

To identify blocks for migration, RheoServe performs a global ranking based on historical attention scores maintained by the lightweight-attention recorder. We employ per-sequence length masks for variable-length batches so that invalid positions do not pollute the ranking. Blocks are sorted by score, with high-importance blocks prioritized for GPU residency. Crucially, we maintain a GPU Block Mask that defines the target capacity for GPU-resident blocks. Our approach leverages a key invariant: after global ranking, the number of CPU-resident blocks in the high-score region exactly equals the number of GPU-resident blocks outside it. This guarantees a balanced exchange without complex traversals. We identify these two sets and perform a direct swap, gathering promoted CPU blocks into a contiguous pinned-memory buffer for efficient DMA transfer.

4 Evaluation

Experimental Setup

Model and Evaluation Tasks. We evaluate on GPQA Rein et al. (2023), which comprises graduate-level questions demanding complex reasoning, along with MATH-500 Hendrycks et al. (2021), AIME 2024, and AIME 2025 Art of Problem Solving (2024), which require rigorous multi-step mathematical problem solving. Based on the observed generation-length distribution, we cap GPQA and MATH-500 at 16K tokens and the AIME tasks at 32K. Accuracy experiments use Qwen3-4B Team (2025), whose compact size enables efficient evaluation across all benchmarks while retaining strong reasoning capability. Accuracy evaluations use deterministic decoding ($temperature = 0$) and reported as pass@1. For throughput evaluation, we test Qwen3-4B and Qwen3-8B under the same 32K decoding protocol, using the corresponding SeerAttention-R predictor for both models. We use a 4096-token attention budget because Figure 4 shows that, in most cases, sparse attention at this budget matches the dense baseline.

Baselines. We evaluate along two axes: quality and performance. For quality, we use vLLM to measure dense full-attention scores, and compare sparse accuracy between RheoServe with SeerAttention-R and ArkVale Chen et al. (2024) with its original sparse predictor. For performance, we compare RheoServe with *PCIe Retrieval*, an ArkVale-derived baseline that fully retains its cache-management and KV-retrieval path. Both systems share the same SeerAttention-R predictor and selected blocks, differing only in cache-miss handling: CPU compute offload versus PCIe retrieval. Additionally, we compare against vLLM Kwon et al. (2023) and SGLang Zheng et al. (2024) with host-memory offload enabled: SGLang HiCache and vLLM's native CPU KV-cache offload (`OffloadingConnector`) are both configured with an 80 GB host KV-cache pool, identical to those of RheoServe and ArkVale. Appendix F further discusses our baseline selection.

Platform. Main throughput results (Figures 5 and 6) are obtained on a consumer workstation with an Intel i9-12900K CPU (8 threads), an NVIDIA RTX 3090 (24GB), and 128 GB DDR4 memory. All main-text end-to-end comparisons use this workstation. The bandwidth microbenchmark additionally reports results on an Intel Core Ultra 7 265K platform with 32 GB DDR5 memory. Appendix F validates cross-platform applicability on newer hardware (RTX 5090, DDR5), running in a container provisioned with 16 CPU cores and 90 GB of memory.

End-to-End Performance.

Beyond decoding throughput, we evaluate RheoServe over complete request execution. Table 1 reports end-to-end latency composition and Time To First Token (TTFT). For a fixed 32K sequence length, we vary the prefill ratio from 25% to 75%. Across these profiles, prompt prefill accounts for 0.29–4.10% of total latency, with generation dominating the remainder.

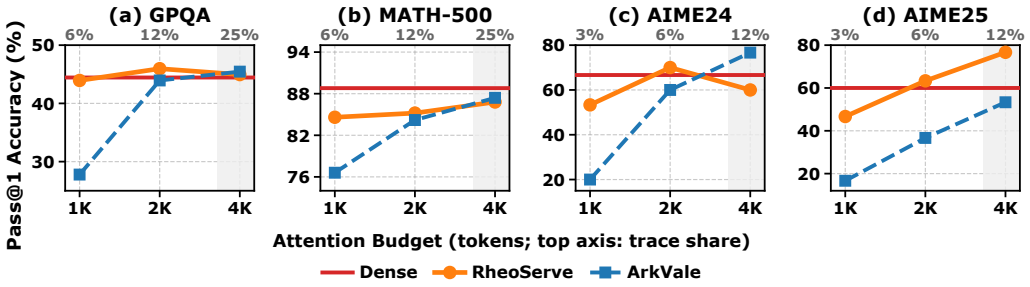


Figure 4: End-to-end pass@1 accuracy on GPQA, MATH, AIME24, and AIME25 across varying token budgets.

(a) End-to-end latency composition (vLLM)

Prefill/Total	25%	50%	75%
Prefill latency	0.29%	1.20%	4.10%
Decode latency	99.71%	98.80%	95.90%

(b) p95 time to first token

Prompt Length	8K	16K	32K
RheoServe p95 (ms)	1370	3478	9926
vLLM p95 (ms)	1270	3355	9929

Table 1: End-to-end latency composition and p95 TTFT.

Regarding TTFT, RheoServe shows a slight overhead at 8K compared to the dense baseline but aligns closely at 32K. Given that generation accounts for the vast majority of end-to-end latency, this minor startup difference has limited impact on overall request completion. These results confirm that RheoServe introduces no significant prefill overhead, allowing the remainder of our evaluation to focus on decoding throughput, the dominant component in long-context serving.

Quality Evaluation

We evaluate end-to-end reasoning accuracy on GPQA, MATH-500, AIME24, and AIME25 using attention budgets of 1K, 2K, and 4K tokens. Both sparse systems use 32-token blocks and the same attention budgets. Since SeerAttention-R is originally designed for 64-token blocks, we train a 32-token-block predictor to match RheoServe's block granularity. ArkVale uses its native sparse-selection method.

Figure 4 shows that as the attention budget increases from 1K to 2K and 4K tokens, both sparse methods progressively approach the dense baseline. At the 4K operating point, sparse attention matches or approaches the dense score in most cases. RheoServe achieves higher accuracy than ArkVale in most evaluated settings, especially at tighter budgets. This advantage is partly attributable to RheoServe's learned block representation: while ArkVale relies on two bounding vectors per block, RheoServe's trained Key Index captures block relevance in a single learned vector.

The gray region marks the 4096-token throughput operating point, which corresponds to 12.5%–25% of the maximum sequence length used in these evaluations. These end-to-end results validate RheoServe's heterogeneous execution path and confirm the correctness of our LSE-based CPU–GPU collaborative implementation, as LSE merging introduces no numerical approximation beyond the sparse selection.

Performance Evaluation

To isolate the effect of the offload mechanism, both systems use the same SeerAttention-R checkpoint and block selections under a 4096-token attention budget with a block size of 32, ensuring identical offload-access traffic. The only difference is how cache misses are handled: RheoServe executes attention over missed blocks on the CPU, whereas PCIe Retrieval, built on ArkVale's retrieval pipeline, transfers them to the GPU. Both systems are evaluated via teacher-forced replay over a 32K-token sequence.

For Qwen3-4B, we evaluate both sparse systems across batch sizes 1–16. For Qwen3-8B, RheoServe scales up to BS16, whereas PCIe Retrieval is limited to BS8. This limitation arises because the

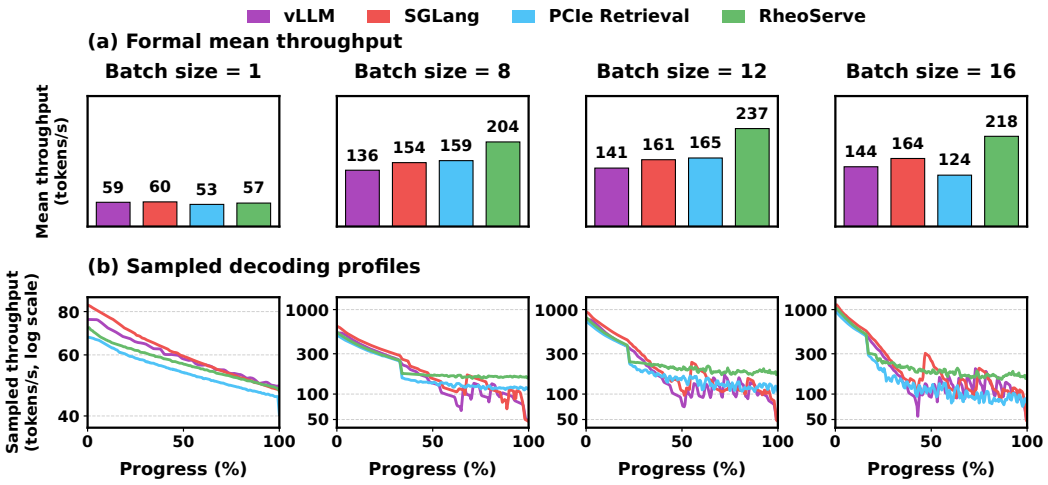


Figure 5: Qwen3-4B throughput across batch sizes. (a) Mean throughput. (b) Decoding profiles versus normalized progress (log scale).

larger model consumes more GPU memory, and PCIe Retrieval additionally requires staging space for recalled host-resident blocks. Beyond BS8, the growing KV cache exhausts this headroom, preventing further scaling. Figure 5 also compares RheoServe with vLLM and SGLang.

Average Throughput. Figure 5 shows that at BS1, vLLM and SGLang lead due to highly optimized GPU kernels under low memory pressure. As batch size grows, however, their KV caches exhaust GPU memory. All systems have host-memory KV offload enabled with the pool capacities specified above. Under this configuration, RheoServe achieves a sustained throughput lead at BS8, BS12, and BS16. At BS12, RheoServe peaks at 236.5 tokens/s, reaching $1.68\times$, $1.47\times$, and $1.43\times$ the throughput of vLLM (141.1), SGLang (161.3), and PCIe Retrieval (165.5), respectively. Unlike the baselines, which move data over PCIe to the GPU for computation, RheoServe moves computation to host-resident data, which proves especially beneficial under heavy memory pressure.

Decoding Behavior Analysis. Figure 5(b) complements these averages with sampled throughput profiles, revealing how each system behaves as decoding progresses. RheoServe dynamically adapts to the aggregate memory footprint. When the KV cache fits within the GPU budget (e.g., at BS1), decoding completes entirely on the GPU. Once the footprint exceeds the budget at larger batch sizes, RheoServe enters sparse mode, dynamically migrating blocks to the CPU and distributing attention computation across both tiers.

As memory pressure builds, throughput drops across all systems. Both sparse systems experience a drop at BS8, while vLLM and SGLang also decline once they begin moving KV states between host and GPU. For the sparse systems, this drop is a one-time transition cost rather than persistent degradation. It reflects the activation of the host-side execution path, where RheoServe exchanges queries and partial results with the CPU, and PCIe Retrieval transfers selected blocks back to the GPU.

After this transition, both sparse systems depend on host-resident data, and the bandwidth advantage of compute offload becomes directly visible. PCIe Retrieval, limited by PCIe transfer bandwidth, trends downward from BS8 onward and declines even more steeply at larger batch sizes, consistent with its lower effective offload bandwidth in Figure 7. Under the same memory pressure, vLLM and SGLang also fluctuate as they move KV between host and GPU. RheoServe, in contrast, sustains a consistently high throughput into the later decoding stages. Only at BS16 does RheoServe itself begin to decline, as the CPU-side share of attention grows, an effect quantified by Table 2 below.

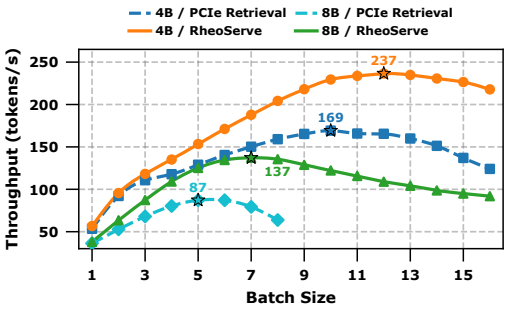


Figure 6: Decoding throughput versus batch size for Qwen3-4B and Qwen3-8B.

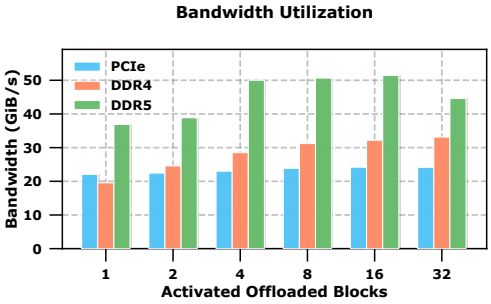


Figure 7: Effective bandwidth of PCIe retrieval and CPU attention on DDR4/DDR5.

Batch Size	GPU ms	CPU ms	CPU miss
4	16.78	2.35	1.29%
8	16.73	8.93	4.77%
12	16.82	25.33	10.47%
16	16.75	52.60	17.62%

Table 2: GPU and CPU attention latency.

Table 2 quantifies this growing CPU-side share by comparing GPU and CPU attention latency at a late decoding step. At BS4 and BS8, CPU attention is shorter than GPU attention and is absorbed by concurrent GPU execution, consistent with the near-linear throughput growth from BS4 to BS8 in Figure 6. As the batch grows, more KV accesses miss the GPU cache and shift work to the CPU. At BS16, CPU attention substantially exceeds GPU attention and becomes the limiting factor. Throughput peaks at BS12 because the benefit of a larger batch outweighs the penalty of a longer step time only up to that point, beyond which step time grows faster than batch size and pulls throughput down. Appendix D formalizes this marginal condition, confirming that throughput rises into BS12 and falls at BS13.

Consistent with the latency analysis in Table 2, throughput in Figure 6 grows linearly from BS4 to BS8 as CPU attention is fully absorbed by concurrent GPU execution. Beyond this region, RheoServe consistently outperforms PCIe Retrieval. On Qwen3-4B, RheoServe peaks at 236.5 tokens/s at BS12, reaching $1.40\times$ the peak of PCIe Retrieval (169.2 tokens/s at BS10). For Qwen3-8B, the same trend holds under greater memory pressure: RheoServe peaks at 137.2 tokens/s at BS7, yielding $1.57\times$ over PCIe Retrieval (87.2 tokens/s at BS5). As the batch size increases further and memory pressure intensifies, the throughput gap widens to $1.76\times$ at 4B BS16 and $2.13\times$ at 8B BS8, while RheoServe's post-peak degradation remains substantially more gradual. Finally, Appendix F validates these scaling trends on new generation platform (RTX 5090 with DDR5).

Offload Bandwidth Evaluation

Finally, we evaluate offload bandwidth across 1–32 activated blocks. The CPU offload path includes query and visit-plan transfers, sparse attention computation, and partial-result transfers, while the PCIe baseline measures contiguous KV-block transfers to GPU memory. Figure 7 shows that RheoServe on dual-channel DDR4 achieves higher effective bandwidth than the PCIe baseline across 2–32 blocks. Supplementary kernel measurements on DDR5 show further bandwidth gains, supporting the potential of compute offload on newer memory platforms.

5 Conclusion

In this paper, we presented RheoServe, a CPU–GPU collaborative sparse-attention framework for efficient LLM decoding. By combining compute offload with adaptive block migration, RheoServe alleviates the PCIe transfer bottleneck. Our results show that commodity hardware with heterogeneous co-execution can support efficient sparse-attention serving under tight GPU memory budgets while preserving reasoning accuracy. We hope this work encourages further research into near-memory compute for resource-efficient LLM inference.

AI USE STATEMENT

In this work, we used generative AI tools for language editing, scripting, and assistance with mathematical derivations and result interpretation. We take responsibility for the final content of this work, including all AI-assisted material.

REPRODUCIBILITY STATEMENT

The main text and appendix provide detailed descriptions of our methodology, experimental setup, and evaluation procedures to support reproducibility. We will publicly release our code immediately upon acceptance.

REFERENCES

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4895–4901, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.298. URL https://aclanthology.org/2023.emnlp-main.298/.

Art of Problem Solving. Aime problems and solutions. https://artofproblemsolving.com/wiki/index.php/AIME_Problems_and_Solutions, 2024.

Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. Arkvale: Efficient generative LLM inference with recallable key-value eviction. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/cd4b49379efac6e84186a3ffce108c37-Abstract-Conference.html.

Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Léon Bottou, Zhihao Jia, and Beidi Chen. Magicpig: LSH sampling for efficient LLM generation. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=ALzTQUgW8a.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR*, abs/2501.12948, 2025. doi: 10.48550/ARXIV.2501.12948. URL https://doi.org/10.48550/arXiv.2501.12948.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence. *CoRR*, abs/2606.19348, 2026. doi: 10.48550/ARXIV.2606.19348. URL https://doi.org/10.48550/arXiv.2606.19348.

Weishu Deng, Yujie Yang, Peiran Du, Lingfeng Xiang, Zhen Lin, Cheng-Zhong Xu, Song Jiang, Hui Lu, and Jia Rao. HGCA: hybrid GPU-CPU attention for long context LLM inference. *CoRR*, abs/2507.03153, 2025. doi: 10.48550/ARXIV.2507.03153. URL https://doi.org/10.48550/arXiv.2507.03153.

Aditya Desai, Shuo Yang, Alejandro Cuadron, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Hashattention: Semantic sparsity for faster inference. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/desai25a.html.

Yizhao Gao, Shuming Guo, Shijie Cao, Yuqing Xia, Yu Cheng, Lei Wang, Lingxiao Ma, Yutao Sun, Tianzhu Ye, Li Dong, Hayden Kwok-Hay So, Yu Hua, Ting Cao, Fan Yang, and Mao Yang. Seerattention-r: Sparse attention adaptation for long reasoning. volume abs/2506.08889, 2025. doi: 10.48550/ARXIV.2506.08889. URL https://doi.org/10.48550/arXiv.2506.08889.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with

the MATH dataset. In Joaquin Vanschoren and Sai-Kit Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/be83ab3ecd0db773eb2dc1b0a17836a1-Abstract-round2.html.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (eds.), *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: LLM knows what you are looking for before generation. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/28ab418242603e0f7323e54185d19bde-Abstract-Conference.html.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof q&a benchmark. *CoRR*, abs/2311.12022, 2023. doi: 10.48550/ARXIV.2311.12022. URL https://doi.org/10.48550/arXiv.2311.12022.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomput.*, 568(C), February 2024. ISSN 0925-2312. doi: 10.1016/j.neucom.2023.127063. URL https://doi.org/10.1016/j.neucom.2023.127063.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. QUEST: query-aware sparsity for efficient long-context LLM inference. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 47901–47911. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/tang24l.html.

Qwen Team. Qwen3 technical report. *CoRR*, abs/2505.09388, 2025. doi: 10.48550/ARXIV.2505.09388. URL https://doi.org/10.48550/arXiv.2505.09388.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=NG7sS51zVF.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for LLM inference serving. In Matei Zaharia, Gauri Joshi, and Yingyan (Celine) Lin (eds.), *Proceedings of the Eighth Conference on Machine Learning and Systems, MLSys 2025, Santa Clara, CA, USA, May 12-15, 2025*. OpenReview.net/mlsys.org, 2025. URL https://openreview.net/forum?id=RXPofAsL8F.

Z.ai. GLM-5.2: Built for long-horizon tasks. Technical blog, https://z.ai/blog/glm-5.2, 2026. Accessed: 2026-07-29.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark W. Barrett, Zhangyang Wang, and Beidi Chen. H2O: heavy-hitter oracle for efficient generative inference of large language models. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/6ceefa7b15572587b78ecfcebb2827f8-Abstract-Conference.html.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In Amir Globersons, Lester

Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

Qihui Zhou, Peiqi Yin, Pengfei Zuo, Chao Wang, and James Cheng. Sparseserve: Unlocking parallelism for dynamic sparse attention in long-context llm serving. In *Proceedings of the 40th ACM International Conference on Supercomputing*, ICS '26, pp. 488–501, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400725227. doi: 10.1145/3797905.3800528. URL https://doi.org/10.1145/3797905.3800528.

A TOWARD MEMORY-SIDE ATTENTION

B FROM KV STORAGE TO KV EXECUTION

The broader goal of RheoServe is to treat the KV cache as a distributed, executable resource rather than passive state attached to a single accelerator. In this framework, logical context is decoupled from physical placement: the attention policy decides which blocks matter, the runtime decides where they reside, and computation follows the data. GPU memory provides the fast working set for hot blocks, while host memory supplies additional capacity and, equally important, local bandwidth and compute for cold blocks. This shifts the design focus from moving KV data across the accelerator link to computing attention results locally where the data resides.

This decoupling also reduces the hardware cost of scaling context length. When KV capacity becomes the primary bottleneck, adding cheaper external memory is far more cost-effective than adding expensive GPUs merely to expand VRAM. Utilizing the existing host DDR is the most direct approach. Looking ahead, dedicated memory-expansion modules equipped with lightweight compute could further extend this heterogeneous design. In such a system, memory capacity can be scaled independently to hold more historical KV state, while the GPU resources remain fully dedicated to core computation. This allows deployments to support extremely long contexts without forcing all state to reside on the accelerator.

However, external capacity is only useful if it can be accessed without creating a new bottleneck. If every selected cold block must still be transferred to the GPU for attention, offloading merely shifts the constraint from capacity to interconnect bandwidth. Table 3 makes this distinction explicit by comparing local memory bandwidth with accelerator-link bandwidth. Increasing the bandwidth of a memory tier does not automatically increase the throughput across the link to the accelerator.

Technology	**Configuration**	**Transfer Rate**	**Bandwidth**
Memory and Near-Memory Storage			
DDR4	Dual channel	3200 MT/s	51.2 GB/s
DDR5	Dual channel	4800 MT/s	76.8 GB/s
DDR5	Dual channel	6400 MT/s	102.4 GB/s
NVMe SSD	PCIe Gen5 x4	–	14.9 GB/s
HBF	8/16-high NAND, UCIe	–	400–3000 GB/s[†]
Interconnect Baseline			
PCIe 4.0	x16 lanes	16 GT/s	31.5 GB/s
PCIe 5.0	x16 lanes	32 GT/s	63.0 GB/s

Table 3: Local memory bandwidth and accelerator-link bandwidth.

[†]Reported HBF specification range for Grades 1–3, approximately 0.4–3.0 TB/s. Source: SK hynix FMS 2026 announcement, August 4, 2026. `https://news.skhynix.com/en/hbf-at-fms-2026/`

Bandwidth calculation: For dual-channel DDR, bandwidth is the transfer rate multiplied by 8 bytes per transfer and by two channels (e.g., DDR5-4800 provides $4800 \times 8 \times 2 \approx 76.8$ GB/s). For PCIe, bandwidth is the per-lane transfer rate multiplied by the number of lanes, divided by eight, and adjusted for encoding efficiency (e.g., PCIe 4.0 x16 provides $16 \times 16 \times 0.125 \times 128/130 \approx 31.5$ GB/s).

RheoServe addresses this boundary by processing selected cold KV blocks on the CPU and returning compact partial results to merge with the GPU output. This leverages host memory bandwidth for computation, minimizing PCIe traffic to lightweight queries, partial results, and control signals. In this design, local memory handles computation over resident data, while PCIe is reserved for cross-tier communication and adaptive block migration.

This paradigm also influences physical integration. If a memory-side module can process its KV stream locally, the accelerator link is relieved of latency-critical retrieval traffic. Instead, it serves as an asynchronous channel for state migration and lightweight result exchange. For a given workload, shifting bulk data movement off the critical path relaxes the strict timing constraints between the capacity modules and the main compute die. This offers more flexibility in the physical arrangement of compute and memory dies, as background migration is inherently less sensitive to interconnect latency than synchronous attention computation. The local memory interface must still sustain its

high-bandwidth KV reads, so communication is not eliminated but redistributed. The extent to which this relaxes routing constraints depends on the specific implementation, the frequency of block migration, and the system's latency targets.

The broader goal is to scale not just storage capacity, but the compute that operates on it. With local compute and appropriate scheduling, memory-side modules could provide both KV storage and a portion of the attention computation, interacting with the accelerator through a common partial-result interface. RheoServe evaluates the CPU–GPU instance of this organization; extending it to dedicated memory or storage modules, and quantifying the resulting cost and physical-design benefits, remain future work. Ultimately, longer contexts benefit from keeping hot, latency-critical data in accelerator memory while storing cold data in external tiers. By computing contributions where the KV resides, systems can avoid repeatedly moving data across the interconnect and scale context length more gracefully.

C RheoServe Attention Pipeline

Algorithm 1 presents RheoServe's collaborative decoding workflow. It establishes a parallel execution path where the GPU handles hot blocks while the CPU concurrently processes cold blocks identified by the Key Index. The pipeline manages data synchronization and merges partial attention scores via LSE reduction.

Algorithm 1 RheoServe per-layer decode step (GPU + CPU)

Input: query Q, current-token key/value K, V, cache state
Output: attention output O
$Q, K \leftarrow \textsc{ApplyRoPE}(Q, K)$
if CPU path is enabled **then**
    Copy Q into a CPU query buffer
end if
Append (K, V) into the current GPU block for each active slot
if a block boundary is reached **then**
    Mirror the finished GPU block into the CPU pool
    Allocate the next logical block
end if
Build a per-layer CPU visit plan using the Key Index
Compute attention over blocks present on the GPU
if the CPU visit plan is non-empty **then**
    Compute attention over selected CPU-only blocks
    Merge GPU and CPU partial results by LSE
end if
Return O

D Why the Throughput Peaks after the CPU–GPU Timing Crossover

As shown in Figure 8 and Table 2, RheoServe reaches its highest measured throughput at BS12. At this operating point, the CPU attention latency (25.33 ms) already exceeds the GPU attention latency (16.82 ms), indicating that CPU–GPU overlap can no longer fully hide the CPU-side computation. However, this timing crossover does not inherently force an immediate throughput drop. Instead, the throughput peak is determined by whether the marginal output gain from an additional request still outweighs the extra step time it introduces.

The scaling curve illustrates this distinction clearly. In the low-batch regime, CPU attention is faster than GPU attention, allowing much of the CPU work to be overlapped with GPU execution. From BS4 to BS8, throughput rises almost linearly from 135.10 to 204.25 tokens/s, with an average gain of 17.29 tokens/s per added request. Here, slope refers to the rise-to-run ratio of each plotted segment in Figure 8(a), using its fixed axis scales and aspect ratio. As the batch grows larger, the slope gradually decreases—falling to 0.58 over BS8–9, 0.48 over BS9–10, and 0.18 over BS10–11—reflecting diminishing returns. Crucially, the slope over BS11–12 remains positive at approximately

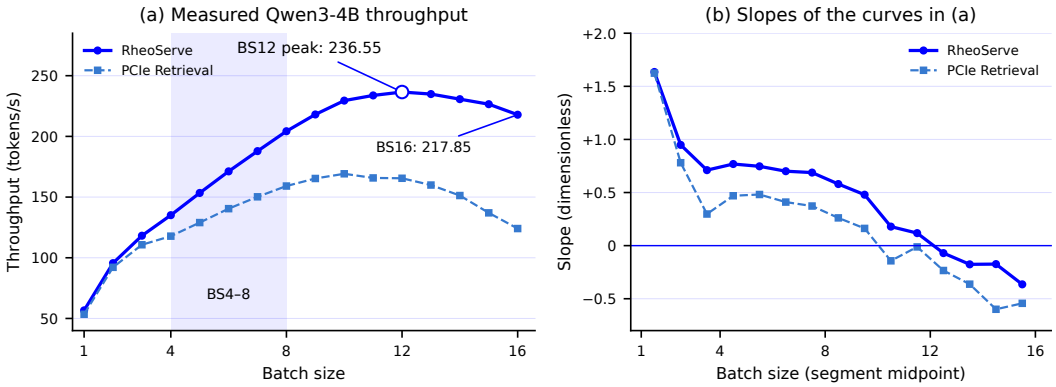


Figure 8: Qwen3-4B throughput versus batch size. (a) Mean throughput; the shaded region highlights the near-linear growth at BS4–8. (b) Slopes of the throughput segments in (a), plotted at the corresponding batch-size midpoints.

$+0.118$, corresponding to a gain of 2.80 tokens/s. Thus, throughput continues to increase even though CPU attention is already the slower component. The slope turns negative only over BS12–13, reaching approximately -0.070 (a loss of 1.68 tokens/s). The measured peak therefore occurs at BS12: not when the CPU first becomes slower than the GPU, but when the marginal benefit of another request can no longer offset its marginal cost.

We can formalize this behavior through a marginal-cost analysis of batch scaling. Let $\bar{t}(B)$ denote the equivalent mean step time in milliseconds, such that the aggregate throughput is $Q(B) = 1000B/\bar{t}(B)$. Let $\Delta t = \bar{t}(B+1) - \bar{t}(B)$ be the additional step time incurred when one more request joins the batch. The throughput difference is then:

$$\begin{aligned} Q(B+1) - Q(B) &= 1000 \left(\frac{B+1}{\bar{t}(B) + \Delta t} - \frac{B}{\bar{t}(B)} \right) \\ &= \frac{1000\big(\bar{t}(B) - B\Delta t\big)}{\bar{t}(B)\big(\bar{t}(B) + \Delta t\big)}. \end{aligned} \tag{4}$$

Since all step times are strictly positive, the sign of $Q(B+1) - Q(B)$ depends solely on the numerator. Therefore,

$$Q(B+1) > Q(B) \quad \Longleftrightarrow \quad \Delta t < \frac{\bar{t}(B)}{B}. \tag{5}$$

Intuitively, throughput continues to grow as long as the additional step time introduced by one more request remains below the current per-request share of the step time. Under this condition, the additional output generated by the larger batch compensates for the exposed CPU latency, so an increase in total step time does not by itself reduce throughput. Equation (5) therefore gives a direct criterion for identifying the throughput peak from the measured end-to-end equivalent step time.

Batch interval	Marginal step time Δt (ms)	Threshold $\bar{t}(B)/B$ (ms)	Throughput change (tokens/s)	Relative change (%)
$11 \to 12$	3.67	4.28	$+2.80$	$+1.20$
$12 \to 13$	4.62	4.23	-1.68	-0.71

Table 4: Marginal-cost condition around the throughput peak. Here $\Delta t = \bar{t}(B+1) - \bar{t}(B)$ is the added equivalent step time, and $\bar{t}(B)/B$ is the threshold below which Δt must remain for throughput to increase.

Table 4 evaluates this condition across the two intervals surrounding the peak. From BS11 to BS12, the equivalent mean step time rises from 47.06 to 50.73 ms, yielding a marginal increase of $\Delta t = 3.67$ ms. Because this is strictly below the BS11 threshold of $47.06/11 = 4.28$ ms, the inequality in Eq. (5) holds. Consequently, throughput still increases by 1.20%, reaching the peak of 236.55 tokens/s. From BS12 to BS13, the step time rises further to 55.35 ms, pushing the marginal increase

to $\Delta t = 4.62$ ms. This exceeds the BS12 threshold of $50.73/12 = 4.23$ ms, causing the inequality to reverse and throughput to fall by 0.71%. This mathematical reversal cleanly identifies BS12 as the empirical throughput peak.

E Evaluation Prompts

The four reasoning-task evaluations in the main paper use a single-turn chat message. The fields in braces below are inserted verbatim from the corresponding benchmark record; no demonstrations or additional system instructions are used. MATH-500 and AIME use the same mathematical-answer template, while GPQA supplies its multiple-choice instruction as part of each question record.

MATH-500 Prompt

```
<|im_start|>user
{problem}
Please reason step by step, and put your final answer within
\boxed{}.<|im_end|>
<|im_start|>assistant
```

AIME 2024 Prompt

```
<|im_start|>user
{problem}
Please reason step by step, and put your final answer within
\boxed{}.<|im_end|>
<|im_start|>assistant
```

AIME 2025 Prompt

```
<|im_start|>user
{problem}
Please reason step by step, and put your final answer within
\boxed{}.<|im_end|>
<|im_start|>assistant
```

GPQA Prompt

Each GPQA record supplies the question, all answer choices, and the following task instruction as a single `question` field. The evaluation places that field directly in the user message.

```
<|im_start|>user
{question with answer choices}
Please reason step-by-step and put your choice letter
without any other text with \boxed{} in the end.<|im_end|>
<|im_start|>assistant
```

For all four tasks, we use deterministic decoding ($temperature = 0$). The maximum sequence length is 16K tokens for GPQA and MATH-500, and 32K tokens for AIME 2024 and AIME 2025, as specified in the experimental setup.

F Supplementary Experimental Results

Baseline Selection

We compare RheoServe against three baselines: ArkVale, vLLM, and SGLang. ArkVale serves as a high-performance representative of PCIe retrieval-based sparse attention, while vLLM and SGLang are widely deployed general-purpose LLM serving frameworks. Together, they cover both optimized sparse-attention execution and established serving systems.

ArkVale's open-source implementation supplies the GPU–CPU KV-cache management, host-block retrieval, data-transfer, and GPU attention-kernel infrastructure needed for a controlled comparison.

In the throughput experiments, RheoServe and the ArkVale-based PCIe Retrieval baseline share the same sparse predictor and block selections, so both systems exercise identical sparse-attention access patterns. This design isolates a single variable: whether the selected host-resident KV blocks are computed directly on the CPU or retrieved over PCIe for GPU computation.

Other hybrid CPU–GPU designs organize the KV cache differently. MagicPIG Chen et al. (2025) also combines CPU and GPU attention, but for long contexts it offloads prompt KV entries to the CPU for LSH-based sampled attention while keeping sink and local tokens on the GPU. In its implementation, newly generated KV entries accumulate on the GPU and participate in dense attention, so the GPU-side attention workload grows as generation proceeds. This differs from continuously refreshing a bounded GPU working set according to evolving block importance (Hot/Cold blocks). HGCA Deng et al. (2025), in turn, keeps recent KV entries in a GPU FIFO buffer and offloads the oldest blocks to the CPU when the buffer fills, applying importance-based sparsification on the CPU. Its residency policy therefore favors recency rather than promoting frequently selected older blocks back into a dynamic GPU working set. In contrast, RheoServe and the ArkVale employ dynamic KV management, handling GPU residency during long generation more comprehensively than the recency or accumulation based policies described above. Building the retrieval baseline on ArkVale further allows us to match the sparse selector and isolate the execution cost of cache misses.

For the dense baselines, we enable vLLM's native CPU KV-cache offload and SGLang's HiCache on the same GPU platform, configuring an 80 GB host KV pool for each framework. RheoServe and PCIe Retrieval use 80 GiB host pools. Both frameworks retain their native offload scheduling and transfer paths, and their host KV buffers reside in pinned memory for efficient transfers. These settings ensure that the dense baselines can also draw on host KV storage once GPU memory becomes constrained, allowing comparison under GPU memory constraints.

QWEN-4B REASONING RESULTS

Table 5 provides the detailed Qwen3-4B reasoning results summarized in the main paper. It compares RheoServe and ArkVale across attention budgets, with full attention as a reference, showing how accuracy varies across tasks as the available KV budget changes.

Task	Dense	1K		2K		4K	
		ArkVale	RheoServe	ArkVale	RheoServe	ArkVale	RheoServe
GPQA	44.44	27.78	**43.94**	43.94	**45.96**	**45.45**	44.95
MATH-500	88.80	76.60	**84.60**	84.20	**85.20**	**87.40**	86.80
AIME24	66.67	20.00	**53.33**	60.00	**70.00**	**76.67**	60.00
AIME25	60.00	16.67	**46.67**	36.67	**63.33**	53.33	**76.67**

Table 5: Qwen3-4B reasoning accuracy (pass@1)

SERVING PROFILES

Figure 9 provides a closer look at the RTX 3090 throughput profiles in Figure 5(b), focusing on the late 50–95% of generation. At BS1, the KV cache fits entirely in GPU memory, allowing vLLM and SGLang to perform slightly better due to their highly optimized pure-GPU kernels. As the batch size increases to 8, 12, and 16, the aggregate KV cache exceeds GPU capacity, forcing all systems to rely on host offload. While the host KV-cache offload in vLLM and SGLang introduces noticeable throughput fluctuations, RheoServe achieves higher late-stage median throughput at BS8, BS12, and BS16, with smaller fluctuations than the other methods at BS12 and BS16. Consequently, over the marked 80–95% progress window, RheoServe sustains a median throughput of 193.5 tokens/s at BS12 and 172.8 tokens/s at BS16, consistently outperforming vLLM, SGLang, and PCIe Retrieval.

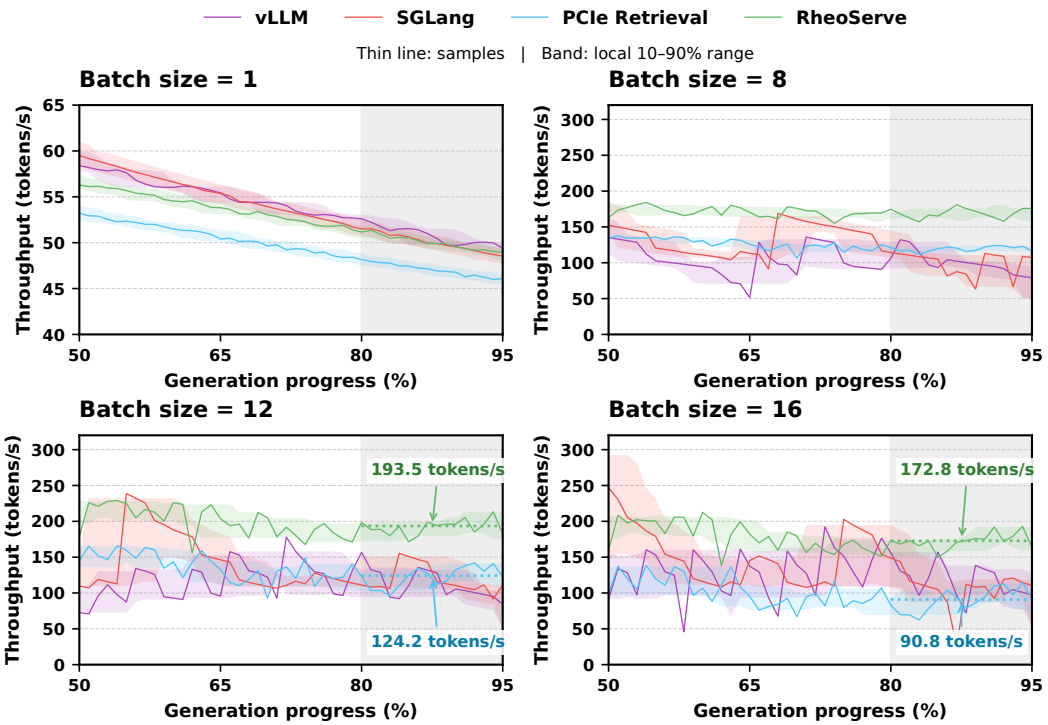


Figure 9: Late-stage throughput profiles with local 10th–90th percentile bands and dotted medians over the shaded 80–95% progress window.

THROUGHPUT RESULTS ON NVIDIA RTX 3090

Table 6 reports the Qwen3-4B and Qwen3-8B throughput results on a workstation with an NVIDIA RTX 3090, an Intel Core i9-12900K, and DDR4 memory, providing the numerical results behind Figure 6. Across the evaluated batch sizes, RheoServe consistently outperforms PCIe Retrieval and achieves higher peak throughput for both models.

(a) Qwen3-4B

Batch size	1	2	3	4	5	6	7	8
PCIe Retrieval	53.47	92.13	110.72	117.81	128.99	140.46	150.22	159.10
RheoServe	**56.65**	**95.59**	**118.17**	**135.10**	**153.40**	**171.18**	**187.87**	**204.25**

Batch size	9	10	11	12	13	14	15	16
PCIe Retrieval	165.32	169.18	165.78	165.49	159.92	151.27	137.00	124.07
RheoServe	**218.06**	**229.48**	**233.74**	**236.55**	**234.87**	**230.67**	**226.53**	**217.85**

(b) Qwen3-8B

Batch size	1	2	3	4	5	6	7	8
PCIe Retrieval	36.32	52.78	67.98	80.31	87.17	87.11	79.29	63.77
RheoServe	**38.01**	**63.46**	**87.32**	**109.19**	**125.20**	**134.57**	**137.19**	**135.52**

Table 6: Decoding throughput (tokens/s) on the RTX 3090 workstation.

THROUGHPUT RESULTS ON NVIDIA RTX 5090

We further evaluate Qwen3-8B on a rented GPU server equipped with an NVIDIA RTX 5090 connected through a PCIe 5.0 $\times$16 link and an Intel Xeon Gold 6459C CPU platform with DDR5 memory. All experiments run inside a container provisioned with 16 virtual CPU cores and a 90 GiB memory limit. Figure 10 and Table 7 report mean decoding throughput across BS1–12.

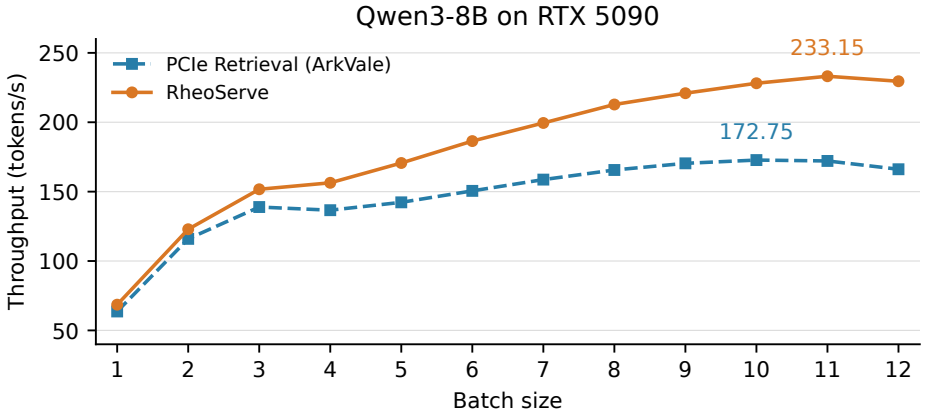


Figure 10: Qwen3-8B throughput on RTX 5090.

Batch size	1	2	3	4	5	6
PCIe Retrieval	63.64	115.96	138.91	136.63	142.29	150.55
RheoServe	**68.54**	**122.90**	**151.70**	**156.36**	**170.64**	**186.43**

Batch size	7	8	9	10	11	12
PCIe Retrieval	158.67	165.60	170.40	172.75	172.12	166.13
RheoServe	**199.48**	**212.79**	**220.93**	**228.10**	**233.15**	**229.54**

Table 7: Qwen3-8B decoding throughput (tokens/s) on RTX 5090.

RheoServe outperforms PCIe Retrieval at every evaluated batch size. Its throughput peaks at 233.15 tokens/s at BS11, compared with 172.75 tokens/s at BS10 for PCIe Retrieval, yielding a $1.35\times$ peak-throughput advantage. At BS12, RheoServe sustains 229.54 tokens/s versus 166.13 tokens/s, a $1.38\times$ advantage. These results show that compute offload retains its throughput advantage on a new GPU platform pairing DDR5 host memory with a PCIe 5.0 interconnect.

Across both DDR4–PCIe 4.0 and DDR5–PCIe 5.0 platforms, compute offload achieves higher peak throughput than PCIe retrieval.

The container allocation limits the CPU resources available to compute host-resident KV blocks. Despite this constraint, RheoServe outperforms the ArkVale-based PCIe Retrieval baseline across all evaluated batch sizes. A dedicated bare-metal server or a high-performance workstation could provide more effective CPU compute and memory bandwidth, potentially reducing the host-side attention latency and extending useful CPU–GPU overlap. Such gains depend on the available bandwidth and execution parallelism and are not measured by this experiment. As discussed in Appendix A, stronger compute resources close to KV storage could further improve performance by processing attention locally.