Zipage: Maintain High Request Concurrency for LLM Reasoning through Compressed PagedAttention

Anonymous ACL submission

Abstract

With reasoning becoming the generative paradigm for large language models (LLMs), the memory bottleneck caused by KV cache during the decoding phase has become a critical factor limiting high-concurrency service. Although existing KV cache eviction methods address the memory issue, most of them are impractical for industrial-grade applications. This paper introduces Compressed PagedAttention, a method that combines token-wise KV cache eviction with PagedAttention. We propose a comprehensive scheduling strategy and support prefix caching and asynchronous compression for Compressed PagedAttention. Based on this, we have developed a high-concurrency LLM inference engine, Zipage. On large-scale mathematical reasoning tasks, Zipage achieves around 95% of the performance of Full KV inference engines while delivering over 2.1× speedup.

1 Introduction

With the advancement of large language models (LLMs), reasoning LLMs have garnered increasing attention from the community (Ke et al., 2025; Li et al., 2025). These models typically perform extensive reasoning before generating answers and have shown remarkable progress in complex domains like code and mathematics. However, as sequence length grows, the memory required for storing the KV cache increases significantly. The core bottleneck of LLM service systems has shifted from computation to having sufficient memory to sustain high-concurrency execution in long sequence scenarios.

Existing KV cache eviction methods can reduce memory usage at the algorithmic level but face fundamental mismatches at the system level. While some methods (Ghadia et al., 2025; Cai et al., 2025; Liao et al., 2025) achieve constant memory usage during decoding, they lack support for advanced techniques like continuous batching and prefix caching, essential features in modern inference engines such as vLLM[1] and SGLang[2]. Consequently, their actual throughput is often lower than engines using a full KV cache. Other methods integrate KV cache eviction into inference engines but rely on **coarse-grained page-wise** eviction, risking the loss of critical information and degrading performance (Hu et al., 2025; Chitty-Venkata et al., 2025). KV-Compress (Rehg, 2024), though employing token-wise eviction, only supports input compression and **disrupts the prefix cache**, significantly increasing prefilling costs.

In this paper, we propose Compressed PagedAttention, a KV cache management approach that combines PagedAttention (Kwon et al., 2023) with **flexible token-wise KV cache eviction across layers and attention heads**. We implemented a high-concurrency inference engine, Zipage[3], based on Compressed PagedAttention, and developed **efficient GPU kernels** to optimize operations during the compression process. Zipage employs a comprehensive request scheduling strategy designed for Compressed PagedAttention and is **compatible with prefix caching**, achieving significant throughput improvements in scenarios where many requests share the same prefix. It also implements **asynchronous compression and decoding** to further enhance throughput.

We evaluated models of various architectures and sizes on reasoning tasks, such as coding and mathematics. Zipage achieved significant throughput gains while maintaining performance close to a Full KV cache engine. Specifically, in mathematical reasoning tasks, Zipage achieves over a 2.1× speedup while retaining approximately 95% of the performance of a Full KV cache engine.

[1] https://github.com/vllm-project/vllm

[2] https://github.com/sgl-project/sglang

[3] Code links will be provided in the final version and will be maintained as a long-term open-source project.

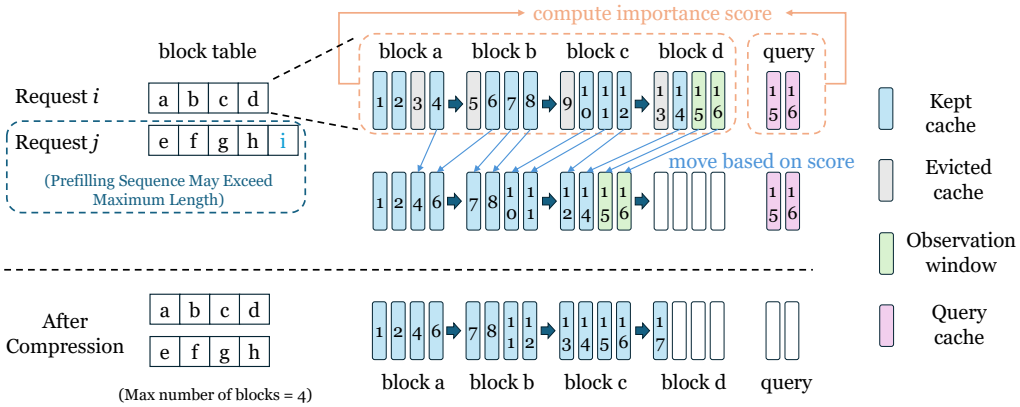


Figure 1: Illustration of Compressed PagedAttention. Here, $N_{\max} = 4, b = 4, w = 2$. The figure depicts two requests requiring compression. After compression, the kept KV cache entries are moved to the first three blocks, while the fourth block is reserved for subsequent decoding. The remaining blocks are released.

2 Related Work

Some methods reduce computational complexity and memory usage by evicting KV cache entries. For example, SnapKV (Li et al., 2024) calculates attention scores to decide which token' KV cache to retain or evict, while PyramidInfer (Yang et al., 2024), PyramidKV (Cai et al., 2024), and Ada-KV (Feng et al., 2024) adjust budgets across heads or layers to improve performance. However, these methods focus on compressing input KV cache, while output KV cache dominates memory in reasoning, limiting concurrency.

Methods such as MorphKV (Ghadia et al., 2025), R-KV (Cai et al., 2025), and G-KV (Liao et al., 2025) perform KV cache eviction during the decoding process, ensuring that the KV cache for each request remains constant. Although these methods significantly improve concurrency, they are not integrated with inference engines and thus cannot be practically applied.

3 Background

PagedAttention (Kwon et al., 2023) is an efficient method for managing the KV cache of Transformer (Vaswani et al., 2017) based LLMs. The LLM serving engine, vLLM, is built on PagedAttention.

Pre-allocated memory. vLLM pre-allocates GPU memory for the KV cache, denoted as $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{L \times N_{\text{total}} \times b \times h_{\text{kv}} \times d}$. Here, L represents the number of layers of LLMs, N_{total} is the total number of blocks, b corresponds to the block size, h_{kv} denotes the number of attention heads, and d is the attention dimension.

KV cache management. vLLM partitions the sequence of each request according to the block size. The KV cache of each block is then written into free blocks in $\mathbf{K}$ and $\mathbf{V}$. vLLM maintains a block table to record the blocks occupied by each request and their corresponding order.

Request scheduling. In vLLM, requests are categorized into two states: waiting and running, and these states are managed using two separate queues. At each decoding step, if sufficient blocks are available for prefilling, vLLM transfers requests from the front of the waiting queue to the running queue and performs prefilling. Otherwise, the requests in the running queue decode the next token. When the last block of a request is filled and new blocks cannot be allocated, the running requests preempt the blocks of the most recently added requests in the running queue. The preempted requests are then moved back to the front of the waiting queue. This scheduling process adheres to a **first-come, first-served (FCFS)** principle.

4 Method

In this work, we propose a novel KV cache management method, Compressed PagedAttention, and develop an LLM serving engine, Zipage.

4.1 Compressed PagedAttention

To address the concurrency challenges, we introduce Compressed PagedAttention, a KV cache management method based on PagedAttention and KV cache eviction strategies. Compressed PagedAttention builds upon PagedAttention by introducing the following key features:

- The number of blocks occupied by each request is capped at $N_{\max}$, except during the

prefilling phase, where the prefilling length may temporarily exceed this limit.

- After each decoding step, if a request occupies N blocks and satisfies $N \geq N_{\max}$ with the last block fully occupied, a compression operation is triggered to evict less important KV cache entries and relocate the retained ones to the first $N_{\max} - 1$ blocks. The $N_{\max}$-th block is reserved for subsequent decoding, while the remaining blocks are released.

Compressed PagedAttention ensures that the memory usage of each request remains within a fixed maximum limit throughout the decoding process, thereby maintaining high concurrency. Figure 1 illustrates the KV cache management mechanism in Compressed PagedAttention.

4.2 The Compression Process Pipeline

In this section, we will further elaborate on the compression process. First, following SnapKV (Li et al., 2024) and MorphKV (Ghadia et al., 2025), we take the query states of the last w tokens in the final block of each request as *observation window*. To accommodate these query states, we pre-allocate memory as $\mathbf{Q} \in \mathbb{R}^{L \times M \times w \times h_q \times d}$, where M represents the maximum concurrency, h_q is the number of attention heads. The maximum concurrency M is subject to the following constraints:

$$\begin{cases} m_{\text{kv}} \times N_{\text{total}} + M \times m_{\text{q}} \leq m_{\text{available}}, \\ M \leq \dfrac{N_{\text{total}}}{N_{\max}}, \\ M > 0, \quad N_{\text{total}} > 0, \end{cases} \tag{1}$$

where $m_{\text{available}}$ denotes the total available memory, m_{kv} represents the memory required for the KV cache of a block, and m_{q} is the memory required to cache the query states of a request. This is a linear programming problem, and the maximum value of M is achieved when $M = \lfloor \frac{m_{\text{available}}}{m_{\text{kv}} \times N_{\max} + m_{\text{q}}} \rfloor$, at which point $N_{\text{total}} = \lfloor \frac{m_{\text{available}}}{m_{\text{kv}} + m_{\text{q}}/N_{\max}} \rfloor$.

When compression is triggered, a scoring function $\phi(\mathbf{Q}, \mathbf{K}, \mathcal{I})$ is employed to assign a score for the KV cache entries of requests that require compression. Here, $\mathcal{I}$ represents additional information, such as the block tables and query slots indexes for these requests. In its basic form , this scoring function involves computing attention scores between the query states in $\mathbf{Q}$ and the key states in $\mathbf{K}$. Furthermore, R-KV (Cai et al., 2025) introduces a redundancy score to evaluate the redundancy of the KV cache, while G-KV (Liao et al., 2025) incorporates a global score to aggregate historical attention scores, providing a better assessment of long-term importance. We integrate these methods into our framework and **implement kernel-level optimizations specifically tailored for the paged KV cache.** Detailed algorithm and experimental results can be found in Appendices C.2, C.3 and C.5.

After obtaining the final scores, we assign a score of $+\infty$ to the entries within the observation window to ensure they are always retained. Subsequently, the top-k KV cache entries with the highest scores are retained, where $k = (N_{\max} - 1) \times b$, referred to as the KV cache budget. Compression is then performed by reorganizing the retained KV cache entries such that their placement in $\mathbf{K}$ and $\mathbf{V}$ becomes compact and contiguous in a page. The full compression algorithm is described in Appendix C.6.

Additionally, although the raw redundancy score from R-KV significantly improves the performance, its computational complexity is $\mathcal{O}(N^2 \times b^2)$, becoming the primary bottleneck in the compression process. To address this issue, we propose a novel **lightning redundancy score** with a reduced computational complexity of $\mathcal{O}(N \times b^2)$, which not only significantly accelerates the compression but also achieves better performance than the raw redundancy score. Detailed descriptions and experiments are provided in Appendix C.7.

4.3 Hybrid Scheduling

To implement a LLM inference engine, a scheduling strategy tailored to Compressed PagedAttention is also crucial. For Compressed PagedAttention, each request requires the allocation of query slots for compression. The number of requests that can be allocated query slots is constrained by the maximum concurrency M. **The simplest scheduling strategy is to restrict the concurrency to no more than M.** Since some requests may occupy more than $N_{\max}$ blocks, it is possible for requests with fewer than $N_{\max}$ blocks to become blocked when attempting to allocate new blocks, even if the concurrency does not exceed M. For requests occupying more than $N_{\max}$ blocks, no additional blocks need to be allocated, and the extra blocks are released after the first compression. At this point, the blocked requests can resume decoding. This scheduling strategy, therefore, enables scheduling **without preemption**, and we refer to it as **con-**

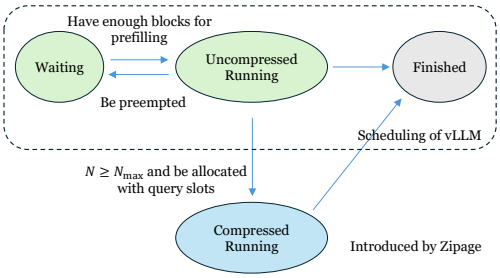


Figure 2: State transition diagram of requests under hybrid scheduling.

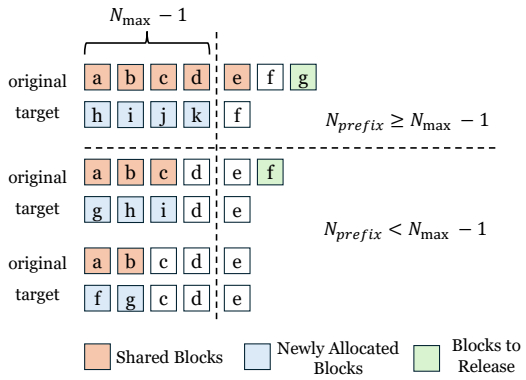


Figure 3: Illustration of block allocation and release strategies for prefix cache.

strained scheduling.

Although constrained scheduling is simple and avoids preemption, this strategy may lead to underutilization of KV cache blocks. When there are a large number of requests with short inputs, the number of blocks occupied by these requests is less than $N_{\max}$, resulting in significant block idleness due to the concurrency limit. This underutilization becomes more frequent in scenarios where only brief responses are required.

To fully utilize these blocks and enhance concurrency, we propose a **hybrid scheduling** strategy. Specifically, the rules of hybrid scheduling can be summarized as follows:

- Only the first M requests in the running queue are eligible for query slots allocation.
- Requests occupying fewer than $N_{\max}$ blocks or with fewer than $b - w$ tokens in the last block can decoding without being assigned query slots. Such requests from the waiting queue can be moved to the running queue for prefilling when sufficient blocks are available, even without query slot allocation. However, requests in the running queue without query slots will be blocked once they no longer meet these conditions.
- When query slots are released, they are priorially allocated to the foremost requests in the running queue that lack assigned query slots.
- If a request in the running queue attempts to allocate a new block but no free blocks are available, preemption is triggered. Priority is given to offloading the last request without assigned query slots. Once all such requests are offloaded, the system **reverts to constrained scheduling**.

Under the hybrid scheduling strategy, the maximum concurrency is no longer constrained by M. The states of requests are illustrated in Figure 2. Requests that have already undergone compression can continue to run without preemption until completion, while uncompressed requests may be subject to preemption. Although the preempted requests discussed in this section are limited to those without assigned query slots, this rule will change in §4.4, where all uncompressed requests, including those with assigned query slots, may be offloaded.

4.4 Shared Prefix Cache for Compressed PagedAttention

Shared prefix cache is a key technique in inference engines. When multiple requests have the same prefix, the KV cache of the prefix can be shared across these requests. This approach reduces both memory usage and the computational overhead of prefilling.

In Compressed PagedAttention, the compression process will disrupt the shared prefix structure and different requests may retain different subsets of KV cache entries, making sharing infeasible. To resolve this, we modify the compression strategy: instead of rearranging KV cache entries within allocated blocks, compression is redirected to a set of target blocks, determined as follows:

- Prefix caching is shared across requests at the block level. Each block tracks the number of requests referencing it. Blocks with a reference count greater than 1 are considered shared.
- If the number of shared blocks for a request, denoted as N_{prefix}, is greater than or equal to $N_{\max} - 1$, we allocate $N_{\max} - 1$ new blocks as taget blocks. The KV cache of the request

is then compressed into these newly allocated blocks.

- If $N_{\text{prefix}} < N_{\max} - 1$, we allocate N_{prefix} new blocks and reuses $N_{\max} - 1 - N_{\text{prefix}}$ blocks already allocated to the request. These combined blocks are used as the target blocks for compression.

With this adjustment, shared prefixes are preserved after compression. Figure 3 illustrates examples of block tables before and after compression. As the compression is completed, the reference count for each shared block is decremented by 1. If the reference count drops to 1, the block is no longer considered a shared block.

Finally, as discussed in §4.3, under constrained scheduling, requests that attempt to allocate new blocks without availability would simply be blocked without releasing any blocks. However, with shared prefixes, new blocks may need to be allocated before compression, meaning requests occupying more than $N_{\max}$ blocks could also face blocking, **potentially leading to deadlocks**. To resolve this, preemption must be applied when prefix sharing is enabled. In such cases, the last **uncompressed request** will be preempted. Although this may occasionally deviate from the first-come, first-served principle, it still prevents prolonged request starvation.

4.5 Asynchronous Decoding and Compression

The previous section introduced the core concepts of Zipage. Here, we evaluate Zipage's performance on reasoning tasks. We measured the average time per step and its proportion of the total time spent on prefilling, decoding, and compression. As illustrated in Figure 4, decoding dominates the overall time consumption in reasoning tasks, while compression accounts for about 10% of the total time. Additionally, the time required for each compression step is approximately $40\% - 70\%$ of that for a decoding step.

We also observe that requests requiring compression constitute less than 1% of the total running requests during each compression operation. Assuming the prefilling length and entry time of each request into the running queue are random, the theoretical proportion of requests needing compression at each step is approximately $\frac{1}{b}$ of the total running requests (where the block size b is 256 in our experiments).

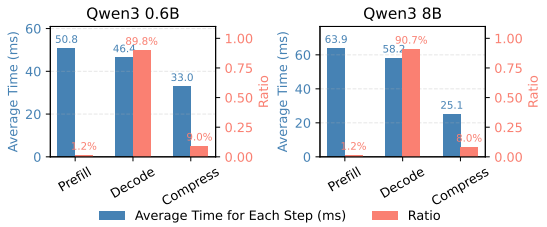


Figure 4: Average time per step and ratio during inference with Qwen3 0.6B and Qwen3 8B on AMC 23 under non-asynchronous compression settings.

This shows that only a small fraction of requests require compression in each step. If compression and decoding are executed sequentially, many requests that do not require compression will be unnecessarily delayed, waiting for the compression of a few requests to finish. Additionally, the small batch size of compression fails to fully utilize the GPU's computational resources, significantly lowering GPU efficiency.

To resolve this, we enable asynchronous execution of compression and decoding. Requests ready for decoding proceed without waiting for compression to finish, while those requiring compression rejoin subsequent decoding steps once asynchronous compression is complete. This design significantly improves GPU utilization and overall throughput.

5 Experiments

5.1 Experimental Setup

We conducted experiments using the Qwen3 series models (0.8B, 8B, 14B, and 32B) (Yang et al., 2025) and DeepSeek-R1 Distill Llama 8B (referred to as DS Llama 8B) model (Guo et al., 2025). We adopt an offline inference manner for evaluation. Except for Qwen3 32B, which runs on 2 A100 GPUs using tensor parallelism, all other experiments are conducted on a single A100 GPU. The block size b was fixed at 256, and the window size w was set to 16. We experimented with larger window sizes, the performance showed almost no difference or even worse, but the memory required to store the queries increased significantly.

To evaluate efficiency, we use two metrics: time per output token (TPOT) and tokens per second (TPS). TPOT is calculated as the total time from the generation of the first token to the last token for a request, divided by the total number of tokens generated for that request. TPS is defined as the total number of tokens generated across all requests, divided by the total time from the start of the first

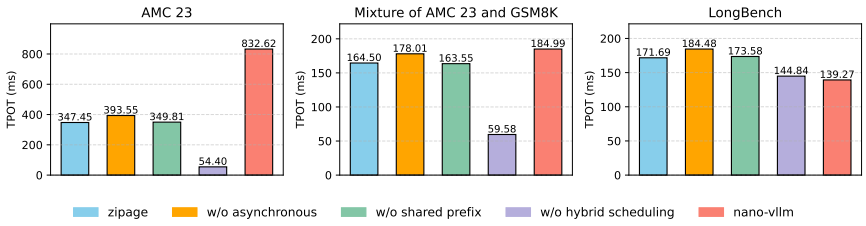


Figure 5: Comparison of average TPOT (ms) of all requests across different configurations on three workloads.

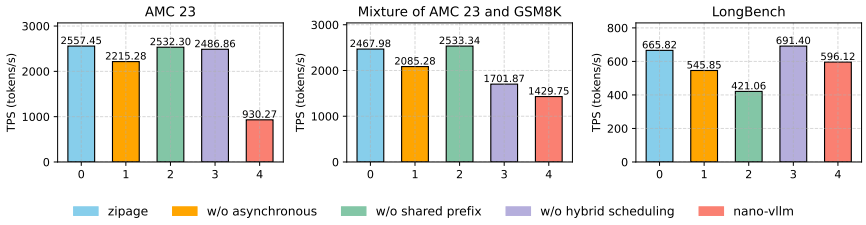


Figure 6: Comparison of TPS (tokens/s) across different configurations on three workloads.

request to the completion of the last. Model performance is assessed using pass@1 as the evaluation metric (Chen et al., 2021). Evaluation settings and benchmark details are provided in Appendix B.

5.2 Efficiency Analysis

To evaluate the efficiency of Zipage, we selected three distinct workload types. The first is the mathematical benchmark AMC 23[4], characterized by short inputs and long outputs. GSM8K[5], by contrast, is a simpler mathematical benchmark with both short inputs and short outputs. For mixed workloads, we combined AMC 23 and GSM8K. Lastly, we selected the MultiFieldQA task from LongBench (Bai et al., 2024) as a representative workload with long inputs and short outputs. The KV cache budget fixed at 2048.

In addition to evaluations using Zipage, ablation studies were conducted on three techniques: asynchronous compression, hybrid scheduling, and prefix sharing. Furthermore, comparisons were made with Nano-vLLM[6], a lightweight implementation of PagedAttention.

Figure 5 shows the TPOT of Qwen3 8B under different configurations. TPOT decreases significantly when hybrid scheduling is disabled, as requests are rarely preempted or blocked, allowing uninterrupted decoding until completion. In contrast, for Zipage with hybrid scheduling or Nano-vLLM, the re-queuing time after preemption can dominate the overall request processing time. **Thus, the TPOT metric becomes less meaningful. Our subsequent analysis will focus primarily on the TPS metric.**

Figure 6 shows the TPS of Qwen3 8B under various configurations. Disabling asynchronous compression consistently lowers TPS across all workloads, underscoring its acceleration benefits in all scenarios. Hybrid scheduling proves advantageous in mixed workloads dominated by short-input and short-output requests, as it improves concurrency. Prefix caching significantly speeds up LongBench due to the presence of long shared prefixes in this workload. Zipage outperforms Nano-vLLM in the TPS metric, with its advantage becoming more evident in scenarios requiring longer outputs. Additional details, including the number of running and waiting requests during inference and block utilization rates, are provided in Appendices D and E.

Figure 7 (a) shows the real-time throughput during inference, calculated as the number of tokens decoded per step divided by the decoding time per step. Nano-vLLM exhibits periodic throughput fluctuations due to offloading requests as sequences lengthen. When a long request completes, the offloaded requests rejoin the running queue, temporarily boosting throughput. In contrast, Zipage maintains consistently high throughput, although asynchronous compression, which competes with decoding for GPU resources, introduces some fluctuations. Figure 7 (b) illustrates the time per step, while Figure 7 (c) compares the average

[4]https://huggingface.co/datasets/math-ai/amc23
[5]https://huggingface.co/datasets/openai/gsm8k
[6]https://github.com/GeeeekExplorer/nano-vllm

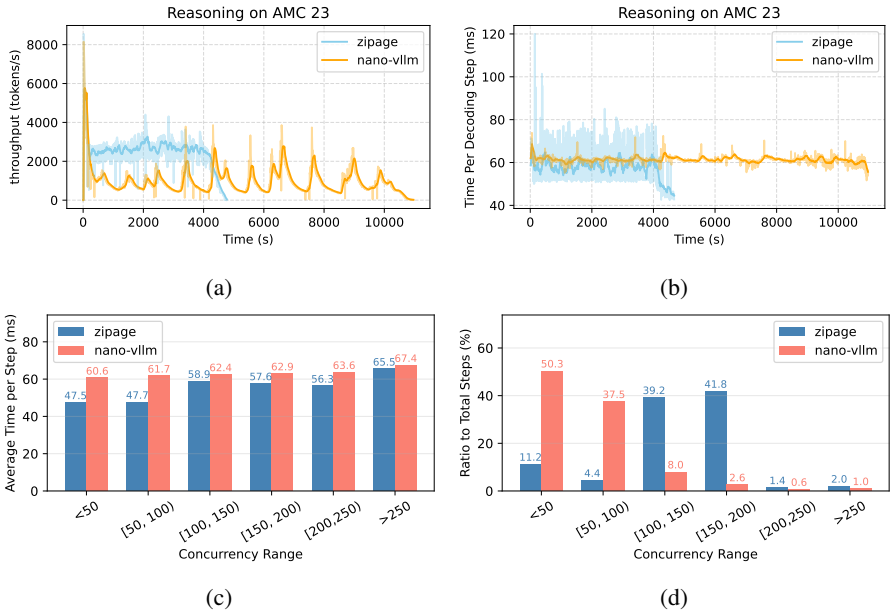


Figure 7: The figure shows Qwen3 8B's performance using Zipage or Nano-vLLM on AMC 23, including:(a) real-time throughput, (b) per-step real-time decoding time, (c) average per-step time at different concurrency range, (d) and the ratio of steps to total steps under different concurrency range.

execution time per step across different concurrency ranges, showing that Zipage achieves shorter times at the same concurrency levels. Figure 7 (d) reveals the proportion of steps across various concurrency ranges, with Zipage primarily operating within the high-concurrency range of [100, 200), whereas Nano-vLLM operates mostly below 100. Additional experimental details for models of different scales are provided in Appendix F.

5.3 Comparison with Other Frameworks

In this section, we compare Zipage with other text generation frameworks. The baselines include HuggingFace generation[7] (HF-Gen) , a Full KV generation framework, as well as MorphKV, R-KV, and G-KV, which incorporate KV cache eviction during decoding to maintain a constant memory. However, none of these methods support advanced techniques such as continuous batching. We also evaluate the inference engines vLLM (v 0.13.0) and Nano-vLLM. vLLM is a highly optimized inference engine for industrial-grade applications.

The evaluation is performed on the AMC 23. For methods that do not support continuous batching, a step size of 5 is used to search for the maximum batch size. For Zipage and other baselines that support KV cache eviction, the KV cache budget is fixed at 2048.

[7]https://huggingface.co/docs/transformers/index

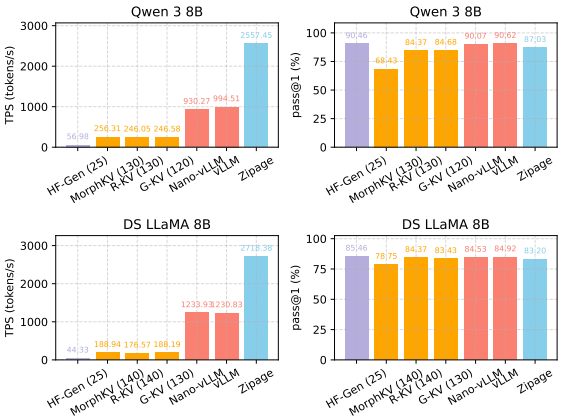
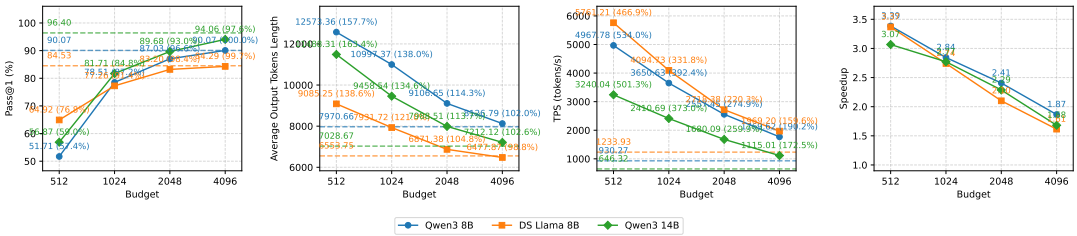
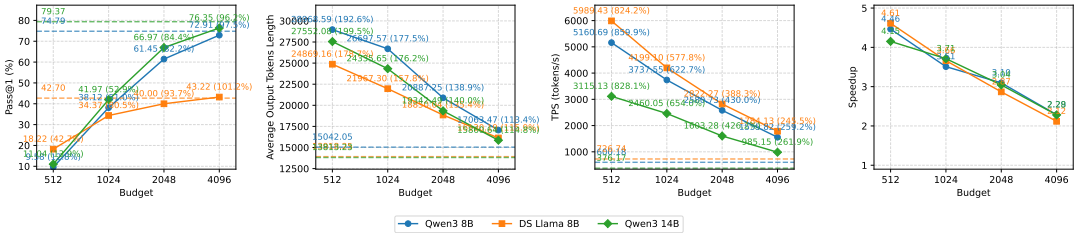


Figure 8: Comparison of TPS and pass@1 performance across different methods for Qwen3 8B and DS LLaMA 8B. The numbers in parentheses indicate the maximum batch size.

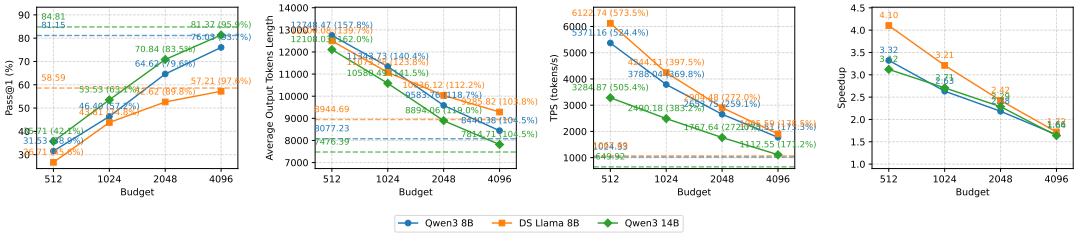
As illustrated in Figure 8, in terms of TPS, methods supporting KV cache eviction demonstrate significant improvements over HF-Gen, primarily due to their capability to handle larger batch sizes. However, these methods, lacking features such as continuous batching, produce a substantial number of padding tokens, causing their TPS to fall below that of inference engines like vLLM and Nano-vLLM. In contrast, Zipage achieves more than double the TPS of both vLLM and Nano-vLLM. Regarding pass@1 performance, methods utilizing KV cache eviction, including Zipage, deliver results compara-



(a) AMC 23



(b) AIME 24



(c) LiveCodeBench

Figure 9: Evaluation results under varying KV cache budgets on (a) AMC 23, (b) AIME 24, and (c) LiveCodeBench. Dashed lines represent the results of full KV, and the percentages following the numerical values indicate the relative performance compared to full KV.

ble to Full KV cache approaches under a 2k budget, with the exception of MorphKV, which exhibits a slight performance gap.

5.4 How to Set KV Cache Budgets?

In this section, we evaluate using different KV cache budgets on two mathematical benchmarks, AMC 23 and AIME 24[8], and one code benchmark, LiveCodeBench (Jain et al., 2024) v1. We report pass@1, average output length, TPS, and speedup, calculated based on the total time required to complete all requests.

For AMC 23, with a budget of 2048, the performance of Zipage reaches around 95% of that of Full KV (Nano-vLLM), while throughput and speedup exceed twice the Full KV baseline. At a budget of 4096, the performance is very close to Full KV. For AIME 24, a 4096 budget achieves about 95% of Full KV performance, with throughput and speedup also exceeding twice the baseline. For code tasks, Zipage achieve around 95% of Full KV performance with a 4096 budget, with a speedup ratio of approximately 1.6. Additionally, we observe that as the budget decreases, not only does performance decline, but the average output length also increases, which may negatively affect user experience.

[8]https://huggingface.co/datasets/math-ai/aime24

6 Discussion

Zipage currently supports RoPE (Su et al., 2024) and its variants, which encoding positional embeddings directly into the KV cache. Zipage is fully compatible with text-based context management systems for multi-turn conversations, and integrates seamlessly with FlashAttention (Dao et al., 2022) as KV cache eviction does not interfere with the Attention forward process.

7 Conclusion

In this paper, we propose Compressed PagedAttention, which integrates KV cache compression with paged KV cache management. Based on this, we develop the inference engine Zipage, which achieves over 2x speedup while delivering performance close to that of Full KV in reasoning tasks.

Limitations

Since we have not yet implemented the online engine, all evaluations are conducted in the form of offline inference. Therefore, we do not report the Time to First Token (TTFT) metric, as queuing time dominates and renders TTFT less meaningful as a reference. In the future, we plan to implement an online engine in the open-source project and integrate techniques such as chunked prefilling (Agrawal et al., 2023) to optimize TTFT.

We did not compare our approach with methods that integrate KV cache compression into inference engines, as RaaS (Hu et al., 2025) and PagedEviction (Chitty-Venkata et al., 2025) lack publicly available code for such integration. However, our token-wise eviction method may offer advantages in preserving critical information. As for KV-Compress (Rehg, 2024), it only compresses inputs and performs similarly to vLLM in scenarios with long outputs.

Additionally, requests of varying difficulty may require different budgets. Currently, we set the budget to a fixed size. However, $N_{\max}$ can be treated as a unique attribute for each request and adjusted based on the actual sequence length of the request, which might slightly reduce concurrency but could improve overall performance. We plan to incorporate this feature into Zipage in the future.

References

Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. 2023. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*.

Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, and 1 others. 2024. Longbench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pages 3119–3137.

Zefan Cai, Wen Xiao, Hanshi Sun, Cheng Luo, Yikai Zhang, Ke Wan, Yucheng Li, Yeyang Zhou, Li-Wen Chang, Jiuxiang Gu, and 1 others. 2025. R-kv: Redundancy-aware kv cache compression for reasoning models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, and 1 others. 2024. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Yaofo Chen, Zeng You, Shuhai Zhang, Haokun Li, Yirui Li, Yaowei Wang, and Mingkui Tan. 2024. Core context aware transformers for long context language modeling. *arXiv preprint arXiv:2412.12465*.

Krishna Teja Chitty-Venkata, Jie Ye, Xian-He Sun, Anthony Kougkas, Murali Emani, Venkatram Vishwanath, and Bogdan Nicolae. 2025. Pagedeviction: Structured block-wise kv cache pruning for efficient large language model inference. *arXiv preprint arXiv:2509.04377*.

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*, 35:16344–16359.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S Kevin Zhou. 2024. Ada-kv: Optimizing kv cache eviction by adaptive budget allocation for efficient llm inference. *arXiv preprint arXiv:2407.11550*.

Ravi Ghadia, Avinash Kumar, Gaurav Jain, Prashant J Nair, and Poulami Das. 2025. Dialogue without limits: Constant-sized kv caches for extended response in llms. In *Forty-second International Conference on Machine Learning*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Junhao Hu, Wenrui Huang, Weidong Wang, Zhenwen Li, Tiancheng Hu, Zhixia Liu, Xusheng Chen, Tao Xie, and Yizhou Shan. 2025. RaaS: Reasoning-aware attention sparsity for efficient LLM reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 2577–2590, Vienna, Austria. Association for Computational Linguistics.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free eval-

uation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Zixuan Ke, Fangkai Jiao, Yifei Ming, Xuan-Phi Nguyen, Austin Xu, Do Xuan Long, Minzhi Li, Chengwei Qin, Peifeng Wang, Silvio Savarese, and 1 others. 2025. A survey of frontiers in llm reasoning: Inference scaling, learning to reason, and agentic systems. *arXiv preprint arXiv:2504.09037*.

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*, pages 611–626.

Andrew Lee, Melanie Weber, Fernanda Viégas, and Martin Wattenberg. 2025. Shared global and local geometry of language model embeddings. *arXiv preprint arXiv:2503.21073*.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 37:22947–22970.

Zhong-Zhi Li, Duzhen Zhang, Ming-Liang Zhang, Jiaxin Zhang, Zengyan Liu, Yuxuan Yao, Haotian Xu, Junhao Zheng, Pei-Jie Wang, Xiuyi Chen, and 1 others. 2025. From system 1 to system 2: A survey of reasoning large language models. *arXiv preprint arXiv:2502.17419*.

Mengqi Liao, Lu Wang, Chaoyun Zhang, Zekai Shen, Xiaowei Mao, Si Qin, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Huaiyu Wan. 2025. G-kv: Decoding-time kv cache eviction with global attention. *arXiv preprint arXiv:2512.00504*.

Isaac Rehg. 2024. Kv-compress: Paged kv-cache compression with variable compression rates per attention head. *arXiv preprint arXiv:2410.00161*.

Noam Shazeer. 2019. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063.

Shawn Tan, Yikang Shen, Songlin Yang, Aaron Courville, and Rameswar Panda. 2024. Stick-breaking attention. *arXiv e-prints*, pages arXiv–2410.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Dongjie Yang, Xiaodong Han, Yan Gao, Yao Hu, Shilin Zhang, and Hai Zhao. 2024. Pyramidinfer: Pyramid kv cache compression for high-throughput llm inference. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 3258–3270.

A Usage of AI

We employed AI to refine the content based on our original text, with all revisions thoroughly reviewed and verified by our team. The code development was assisted with AI. All code underwent rigorous testing.

B Evaluation Settings and Dataset Details

For all evaluations, the sampling temperature is set to 0.6. The evaluation settings for different benchmarks are detailed in Table 1.

Workloads	Number of Questions	Sample Times	Max Output Length
AMC 23	40	32	16384
AIME 24	30	32	32768
LiveCodeBench v1	400	8	16384
Mixture	40+1319	4	16384
LongBench	150	4	4096

Table 1: Evaluation settings for different workloads or benchmarks.

For GSM8K, we use Qwen3's non-reasoning mode, which reduces output length in most cases, though the model occasionally generates lengthy reasoning. Additionally, for the Mixture of GSM8K and AMC23 workload, the question order is randomized.

C Implementation for Compression Operations and Experiments.

For most operations during the compression process, we implemented specialized GPU kernels using Triton[9]. This section contains algorithm descriptions and experiments for these kernel implementations.

C.1 Cross-Layer Parallel Compression

Since the KV cache compression processes of different layers are independent of each other, compression can be executed in parallel across layers.

[9]https://github.com/triton-lang/triton

All kernels can parallelize at least across the dimensions of batch size, layer, and attention head.

However, the compression process generates intermediate activations. If all layers are compressed simultaneously, it may lead to memory overflow in extreme cases, especially under asynchronous compression settings where memory is shared with prefilling and decoding operations. To address this, we adapt cross-layer compression based on a layer stride l, i.e., compressing the KV cache of l layers at each time. The peak of the activations scales as $\mathcal{O}(n \times l \times h_{\text{q}} \times N \times b \times w)$, where n is the batch size of requests requiring compression, and N is the maximum number of blocks among all requests.

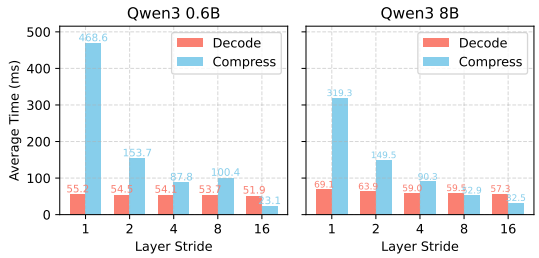


Figure 10: Average time for compression and decoding step under different layer stride.

We inference on AMC 23 and report the average compression and decoding times for different layer strides are shown in Figure 10. As the layer stride increases, both the average decoding time per step and the compression time exhibit a decreasing trend. Notably, the compression time for a layer stride of 16 is approximately $5\% - 10\%$ of that for a layer stride of 1.

Furthermore, we observe that the average compression time for the 0.6B model is longer than that for the 8B model. This is attributed to the higher concurrency in the 0.6B model, resulting in a larger average batch size for each compression step.

Ultimately, we adopt a layer stride of 8 in other experiments, which provides a significant acceleration while maintaining a moderate size for intermediate activation values.

C.2 Paged Attention Score

The scoring function $\phi(\mathbf{Q}, \mathbf{K}, \mathcal{I})$ in its most basic form involves only the computation of attention scores. In this section, we describe how to calculate attention scores using the query states in the query cache $\mathbf{Q}$ and the key states in the key cache $\mathbf{K}$. Since the computations for different requests within a batch, different layers, and different attention heads are independent and executed in parallel, **we illustrate the algorithm with a single request, a single layer, and a single attention head.**

Algorithm 1 Block Attention Logits Computation

Require: Query cache $\mathbf{Q}$, key cache $\mathbf{K}$, block size b, block index i, query slots index j, block table, attention dimension d

Compute query offset p_q in $\mathbf{Q}$ based on query slots index j
Load query states: $\mathbf{Q}_j \leftarrow \mathbf{Q}[p_q] \in \mathbb{R}^{w \times d}$
Get the block id in block table through block index i
Compute key offset p_k in $\mathbf{K}$ using block id
Load key states: $\mathbf{K}_i \leftarrow \mathbf{K}[p_k] \in \mathbb{R}^{b \times d}$
Compute attention logits: $\mathbf{A}' \leftarrow \frac{\mathbf{Q}_j \cdot \mathbf{K}_i^\top}{\sqrt{d}} \in \mathbb{R}^{w \times b}$
if block i is the last block the request **then**
    Construct causal mask $\mathbf{M} \in \mathbb{R}^{w \times b}$ where

$$\mathbf{M}_{u,v} = \begin{cases} -\infty & \text{if } u + b - w > v \\ 0 & \text{otherwise} \end{cases}$$

    Apply mask: $\mathbf{A}' \leftarrow \mathbf{A}' + \mathbf{M}$
end if
Save: $\mathbf{A}[i] \leftarrow \mathbf{A}'$

When computing the attention scores, we first allocate storage space $A \in \mathbb{R}^{N \times w \times b}$. Then, as described in Algorithm 1, we perform the matrix multiplication of query states and key states. Since the attention computation can be further parallelized along the dimension of block numbers, Algorithm 1 outlines the computation for a single block.

It is important to note that, although the matrix multiplication in algorithms is conceptually completed in a single step, in practice, it is further divided into smaller blocks for computation. What's more, *load* in algorithms refers to reading data from the GPU's global memory (Dynamic Random Access Memory, DRAM) into the shared memory or registers (Static Random Access Memory, SRAM), while *save* refers to writing data from the shared memory back to the global memory.

The layout of key states in $\mathbf{K}$ is paged, but the computed logits $\mathbf{A}$ are stored contiguously. So, we can reshape $\mathbf{A}$ into $\mathbb{R}^{w \times (Nb)}$ and apply Softmax along the last dimension to obtain the attention scores $\mathbf{S}' \in \mathbb{R}^{w \times (Nb)}$.

For multi-query (Shazeer, 2019) or group-query

(Ainslie et al., 2023) attention, where a single key head corresponds to multiple query heads, we perform a max-reduce operation for the scores of each key head. Finally, we take an average along the observation window length dimension to obtain $\mathbf{S} \in \mathbb{R}^{N \times b}$.

C.3 Global Score

The global score proposed in G-KV (Liao et al., 2025) combines historical attention scores through decayed max-reduce or sum-reduce, enabling better evaluation of the long-term importance of KV cache entries during eviction. We have integrated the global score into our framework and adapted it for PagedAttention.

First, we need to pre-allocate $\mathbf{F} \in \mathbb{R}^{L \times N_{\text{total}} \times b \times h_{\text{kv}}}$ to store the global score. The size of $\mathbf{F}$ is $\frac{1}{2d}$ of the total size of $\mathbf{K}$ and $\mathbf{V}$. If global score is enabled, equation (1) needs be updated as:

$$\begin{cases} (1 + \frac{1}{2d}) \times m_{\text{kv}} \times N_{\text{total}} + M \times m_{\text{q}} \leq m_{\text{available}}, \\ M \leq \frac{N_{\text{total}}}{N_{\max}}, \\ M > 0, \quad N_{\text{total}} > 0, \end{cases} \tag{2}$$

Algorithm 2 Update Global Score Cache with Attention Scores

Require: Attention score tensor $\mathbf{S}$, global score cache $\mathbf{F}$, block table, number of blocks N, decay factor α, block index i

Load attention scores of i-th block $\mathbf{s}_i \in \mathbb{R}^b$ from $\mathbf{S}$
Compute offset p of i-th block in $\mathbf{F}$ using block id from block table
if request is not compressed **then**
    Save score to cache: $\mathbf{F}[p] \leftarrow \mathbf{s}_i$
else
    if i-th block is not the last block of the sequence **then**
        Load previous global score $\mathbf{f}_i \leftarrow \mathbf{F}[p] \in \mathbb{R}^b$
        Update: $\mathbf{s}_i \leftarrow \max(\alpha \cdot \mathbf{f}_i, \mathbf{s}_i)$
    end if
    Save score to cache: $\mathbf{F}[p] \leftarrow \mathbf{s}_i$
    Overwrite attention score: $\mathbf{S}[i] \leftarrow \mathbf{s}_i$
end if

Based on the attention scores $\mathbf{S}$, we use Algorithm 2 to compute the global scores. The algorithm can be summarized as follows: if a request has not been compressed, there are no historical scores, and we simply store $\mathbf{S}$ in $\mathbf{F}$. If a request has been compressed, all blocks except the last one have historical scores. For these blocks, we take the maximum value between the decayed historical scores and $\mathbf{S}$ as the new score.

α	0	0.4	0.8	0.85	0.9	1
Qwen3 8B	0.6906	0.7375	**0.7718**	0.7468	0.7484	0.7531
DS Llama 8B	0.7484	**0.7656**	0.7643	0.7584	0.7578	0.7515

Table 2: Experimental results with different decay rates α.

We evaluate using a budget of 2048 and global score with different decay rates α on the AMC 23 benchmark, with the experimental results shown in Table 2. The global score shows a significant improvement on Qwen3 8B but provides minimal benefits on DS Llama 8B.

C.4 Pooling at the Sequence Dimension

SnapKV (Li et al., 2024) performs max pooling along the sequence dimension, meaning that tokens near high-scoring tokens are also assigned high scores. This helps preserve more detailed information in context. We have integrated this method into our framework, implementing it using PyTorch[10]'s MaxPool1D interfaces instead of a specialized kernel:

$$\mathbf{S} = \text{MaxPool1D}(\mathbf{S}) \tag{3}$$

Pooling is performed after computing the global score during the compression process.

C.5 Redundancy Score of Key states

R-KV (Cai et al., 2025) introduces redundancy scores to evaluate the degree of redundancy among KV cache entries. Specifically, it calculates the cosine similarity between the key states within a sequence.

Figure 11 illustrates the computation of redundancy scores between the key states of two blocks. The diagonal entries represent the similarity of key states with themselves and are therefore zeroed out. Additionally, for each column, the last similarity score exceeding the threshold p is set to 0, as we prioritize retaining newer tokens when an old token is highly similar to a new token. Finally, the similarity matrix is summed row-wise, normalized

[10] https://pytorch.org/

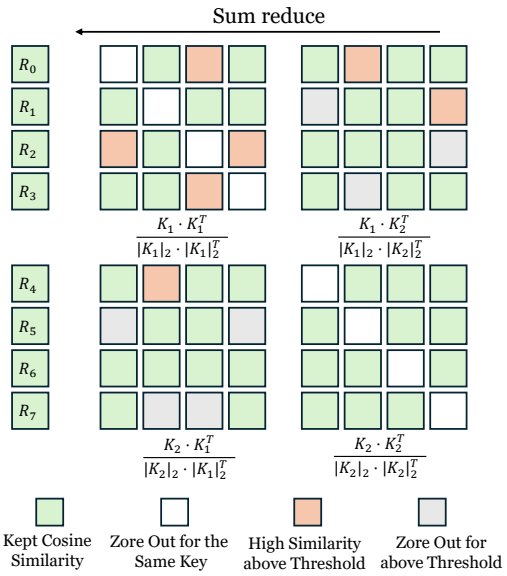


Figure 11: This figure illustrates the computation of the original redundancy scores when $N = 2$ and $b = 4$. Here, $\mathbf{K}_i$ represents the key states of the i-th block. This approach has a computational complexity of $\mathcal{O}(N^2 \times b^2)$ and a memory complexity of $\mathcal{O}(N^2 \times b^2)$.

by the sequence length, and passed through Softmax on the sequence dimension to compute the redundancy score $\mathbf{R}$.

The redundancy score is applied after max pooling. The redundancy score is combined with previous scores using the following formula:

$$\mathbf{S} = \mathbf{S} - \lambda \cdot \mathbf{R} \tag{4}$$

The original implementation of the redundancy score first computes the complete cosine similarity matrix and then applies zeroing out. Its memory complexity is $\mathcal{O}(N^2 \times b^2)$. Assuming a floating-point size of 2 bytes, the actual matrix size is $2 \times n \times l \times h_{\text{kv}} \times N^2 \times b^2$. For a common scenario where $n = 16$, $l = 8$, $h_{\text{kv}} = 8$, and $b = 256$, the cosine similarity matrix size is $128 \times N^2$ MB. When the sequence length is sufficiently large, the memory usage can even reach tens of GB.

Such enormous activations are unacceptable and can easily lead to memory overflow. To address this issue, we implemented the flash redundancy score. Figure 12 illustrates the computation process of the flash redundancy score. We no longer store the complete similarity matrix. Instead, we compute similarities in a block-wise manner, starting from the last block. The computed similarity results are not retained but are directly accumulated into a pre-allocated accumulation accumulator $\mathbf{R}' \in \mathbb{R}^{N \times N \times b}$. To correctly zero out the last high-threshold similarity in each column, we maintain a zero-out tag to track whether the last similarity score exceeding the threshold in each column has been zeroed out. If a column has not been zeroed out, the last value in the block that exceeds the threshold is set to 0, and the corresponding zero-out tag will be set. The detailed process is shown in Algorithm 3. Finally, we perform sequential accumulation to obtain $\sum_{m=0}^{N-1} \mathbf{R}'[:, m]$, and then apply length normalization and Softmax to compute $\mathbf{R}$.

Algorithm 3 Flash Redundancy Score

Require: Key cache $\mathbf{K}$, block table, number of blocks N, threshold p, block size b, block index m

Ensure: Accumulated similarity score $\mathbf{R}'$

Calculate the offset p_m in $\mathbf{K}$ based on the block id of the m-th block
Load key states $\mathbf{K}_m \leftarrow \mathbf{K}[p_m] \in \mathbb{R}^{b \times d}$ of block m
Initialize zero-out tag $\mathbf{z} \in \mathbb{R}^{1 \times b} \leftarrow 0$
for $i = N - 1$ to 0 **do**
    Calculate the offset p_i in $\mathbf{K}$ based on the block id of the i-th block
    Load key states $\mathbf{K}_i \leftarrow \mathbf{K}[p_i] \in \mathbb{R}^{b \times d}$ of block i
    Compute cosine similarity: $\mathbf{C} = \frac{\mathbf{K}_i \cdot \mathbf{K}_m^\top}{\|\mathbf{K}_i\|_2 \cdot \|\mathbf{K}_m\|_2^T} \in \mathbb{R}^{b \times b}$
    if $i = m$ **then**
        Mask diagonal of $\mathbf{C} \leftarrow 0$
    end if
    Identify the last element $> p$ in the column of $\mathbf{C}$, ensure that the corresponding tag in $\mathbf{z}$ for this column is 0, and set this element to 0
    Update the tag corresponding in $\mathbf{z}_m$ to 1 where such element were zero out
    $\mathbf{C}' \in \mathbb{R}^{b \times 1} \leftarrow$ Row-wise accumulate $\mathbf{C}$
    save the result to $\mathbf{R}'[i, m] \leftarrow \mathbf{C}'$
end for

The flash redundancy score reduces the memory complexity to $\mathcal{O}(N^2 \times b)$. By partitioning the matrix multiplication into smaller blocks, the intermediate similarity scores and zero-out tags are temporarily stored using registers and shared memory. Only the accumulated results need to be written to $\mathbf{R}'$ in the global memory. In the previous example, the activation size of the flash redundancy score is approximately $\frac{N^2}{2}$ MB, which is $\frac{1}{256}$ of the original implementation.

An even more aggressive implementation exists,

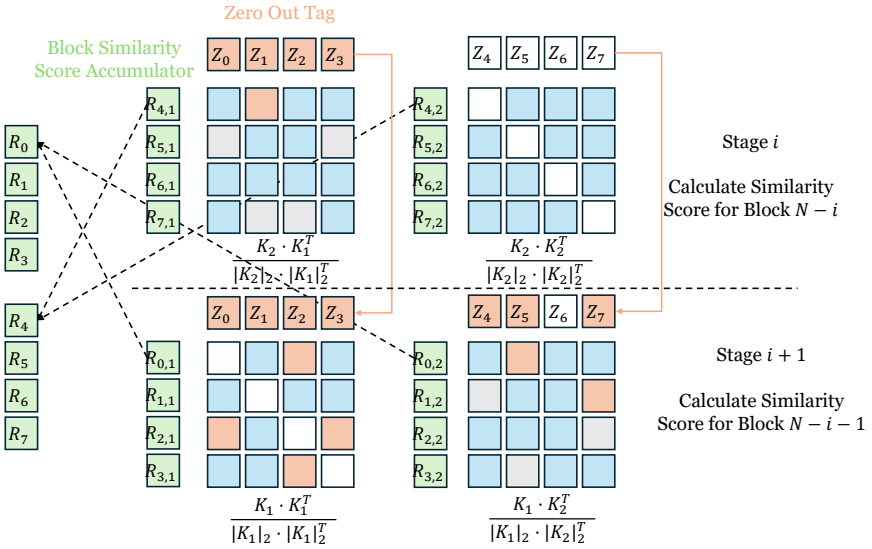


Figure 12: This figure illustrates the computation of flash redundancy score. The green blocks represent data stored in global memory, i.e., activations. The results of other computations, shown in different colors, are temporarily stored using registers or shared memory. This approach has a computational complexity of $\mathcal{O}(N^2 \times b^2)$ and a memory complexity of $\mathcal{O}(N^2 \times b)$.

where the accumulated activations for similarity scores only require $\mathcal{O}(N \times b)$ memory space. This approach involves directly accumulating the results of $\frac{\mathbf{K}_i \cdot (\mathbf{K}_1;\ldots;\mathbf{K}_N)^\top}{|\mathbf{K}_i|_2 \cdot |\mathbf{K}_1;\ldots;\mathbf{K}_N|_2^\top} \in \mathbb{R}^{b \times (bN)}$. However, due to the limited capacity of registers and shared memory, only small-scale matrix operations, such as 16×64, can be performed at a time. Large matrix computations require the kernel to execute additional iterations, which reduces the overall level of parallelism. Our current implementation adopts a balanced approach, trading off between memory usage and parallelism.

C.6 KV Cache Compression

After obtaining the final scores, we first set the scores corresponding to the observation window to $+\infty$. The kernel implementation for this step is relatively straightforward and will not be discussed in detail. Subsequently, we generate a top-k tag $\mathbf{T} \in \mathbb{R}^{N \times b}$ ($k = (N_{\max} - 1) \times b$), where the tag of k KV cache entries with the highest scores along the sequence dimension are marked as 1, while the remaining entries are marked as 0. The top-k tagging is implemented using PyTorch's built-in interfaces. Based on the top-k tag, we reorganize the KV cache placement to ensure that the retained KV cache entries are densely packed in memory.

The algorithm for the compression process is shown in Algorithm 4. Simply put, it is based on the movement of data in memory using two pointers. In total, it requires $(N_{\max} - 1) \times b$ reads and writes to the KV cache.

It should be noted that when using the global score, the historical scores stored in $\mathbf{F}$ also need to be moved correspondingly to correctly match the associated KV cache entries. The process is similar to Algorithm 4, but the amount of data moved each time is reduced from d to 1.

C.7 Lightning Redundancy

The previous sections have introduced all the operations involved in the compression process. We visualized the average execution time of each operation during inference with Qwen3 8B on the AMC 23 benchmark, as shown in Figure 13 (non-asynchronous compression) and Figure 14 (asynchronous compression), with the red bars representing the execution time. It is evident that the computation of redundancy scores is the bottleneck in the compression process, requiring 1-2 orders of magnitude more time than other operations. This is due to its computational complexity of $\mathcal{O}(N^2 \times b^2)$, which is nearly equivalent to the complexity of attention computation during prefilling.

To accelerate the compression process, we propose a novel **lightning redundancy score**. Specifically, since highly similar representations exhibit locality in the sequence space (Lee et al., 2025), meaning that the hidden representation of a token is more similar to those of nearby tokens, this phenomenon may be attributed to the attention mecha-

Algorithm 4 KV Cache Compression

Require: Key and Value cache tensors $\mathbf{K}, \mathbf{V}$, block size b, number of blocks N, top-k tag $\mathbf{F} \in \{0,1\}^{N \times b}$
Ensure: Compressed $\mathbf{K}$, $\mathbf{V}$

Initialize read offset p_r and write offset p_w to the first slot of the first block
$\ell \leftarrow 0, s \leftarrow 0, i \leftarrow 0$
while $i < N$ **do**
  if $\mathbf{F}[i][\ell \bmod b] = 1$ **then**
    Load key vector $\mathbf{k} \leftarrow \mathbf{K}[p_r] \in \mathbb{R}^{1 \times d}$
    Load value vector $\mathbf{v} \leftarrow \mathbf{V}[p_r] \in \mathbb{R}^{1 \times d}$
    Store $\mathbf{k}$ to $\mathbf{K}[p_w]$
    Store $\mathbf{v}$ to $\mathbf{V}[p_w]$
    $s \leftarrow s + 1$
    if $s \bmod b = 0$ **then**
      Move p_w to the first slot of the next block
    else
      Increase p_w to the next slot
    end if
  end if
  $\ell \leftarrow \ell + 1$
  if $\ell \bmod b = 0$ **then**
    Move p_r to the first slot of the next block
    $i \leftarrow i + 1$
  else
    Increase p_r to the next slot
  end if
end while

nism focusing more on tokens in close proximity (Tan et al., 2024; Chen et al., 2024). Based on this observation, we propose computing similarity only between keys within the same block and zeroing out only the last similarity score in each column that exceeds the threshold within the block. We refer to this approach as the lightning redundancy score, and Figure 15 illustrates its calculation process.

The lightning redundancy score reduces the computational complexity to $\mathcal{O}(N \times b^2)$ and the memory complexity to $\mathcal{O}(N \times b)$. As illustrated by the blue bars in Figure 13 and Figure 14, the computation time for the lightning redundancy score is significantly reduced. Figure 16 further presents the average decoding time and compression time under both asynchronous and non-asynchronous compression settings. It is evident that the lightning redundancy score decreases the compression

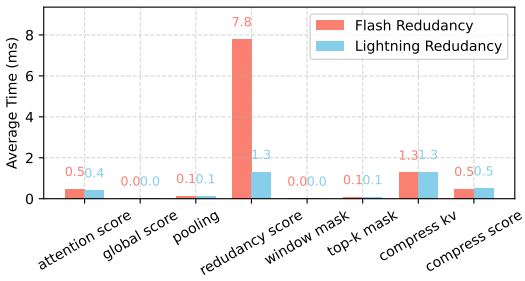


Figure 13: The average execution time of different operations when asynchronous compression is disenabled. A value of 0.0 indicates that the average execution time is less than 0.1 milliseconds.

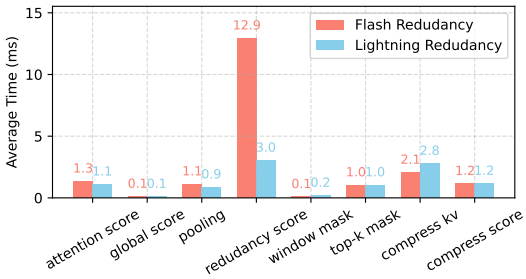


Figure 14: The average execution time of different operations when asynchronous compression is enabled.

time to a level comparable to that of single-step decoding, without impacting the asynchronously executed decoding process. In contrast, the flash redundancy score, due to its substantial computational overhead, intensifies resource contention with the decoding threads, leading to an increase in average decoding time.

λ	0.05	0.1	0.2	0.5	0.9
Flash	80.78	82.96	84.37	**84.53**	**77.81**
Lightning	**83.59**	**84.84**	**84.84**	84.21	75.00

Table 3: Qwen3 8B on AMC 23

Finally, we compare the performance of the two different redundancy scores under various hyperparameters λ. As shown in Table 3 and Table 4, in most cases, the lightning redundancy score achieves even better performance.

C.8 Combining All These Techniques

In this section, we aim to combine the previously discussed methods. G-KV (Liao et al., 2025) has attempted to integrate the global score and the redundancy score. However, due to the use of max normalization for the scores, it required re-tuning

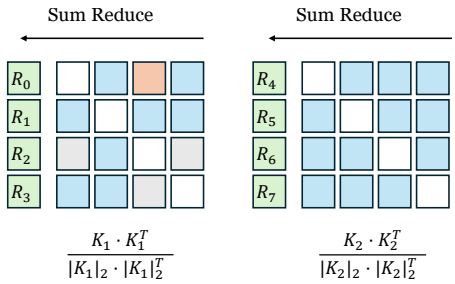


Figure 15: This figure illustrates the computation of lightning redundancy score. This approach has a computational complexity of $\mathcal{O}(N \times b^2)$ and a memory complexity of $\mathcal{O}(N \times b)$.

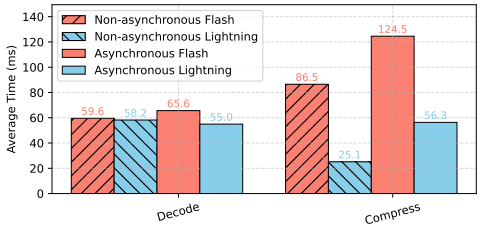


Figure 16: The average decoding time and compression time under both asynchronous and non-asynchronous compression settings.

the hyperparameter λ, making it more sensitive to parameter selection. In our approach, we eliminate the max normalization. Additionally, the redundancy score is ultimately calculated as a distribution via softmax. We observed that this distribution is relatively uniform, especially in the shallow layers, where the differences between scores are minimal. To address this, we introduce a temperature parameter τ for the softmax computation of the redundancy score, which amplifies highly redundant scores.

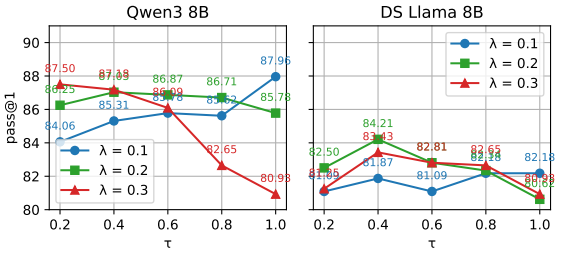


Figure 17: pass@1 performance for Qwen3 8B and DS Llama 8B under diffirent λ and τ.

For the joint search of λ and τ ($\alpha = 0.8$ for global score), the experimental results are shown in Figure 17. The temperature has a significant impact on Qwen3 8B; as λ increases, representing a higher proportion of the global score, lowering the temperature often yields better results. However, for DS Llama 8B, the final performance is less sensitive to the relationship between λ and τ.

λ	0.05	0.1	0.2	0.5	0.9
Flash	**82.96**	82.03	83.59	78.59	**74.21**
Lightning	80.62	**85.15**	**85.00**	**80.78**	73.90

Table 4: DS-Llama 8B on AMC 23

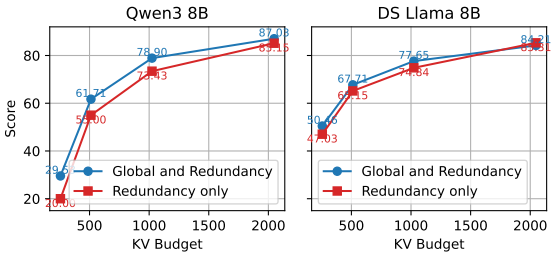


Figure 18: Comparison under different KV budgets for Qwen3 8B and DS Llama 8B, using global and redundancy scores versus redundancy-only scores.

Figure 18 presents the ablation study on the use of the global score. For the Qwen3 8B model, the global score provides significant benefits. However, for DS Llama 8B, when the budget is sufficient (2048), not using the global score can even yield better results.

Model	w/o pooling	pooling once	w/ pooling
Qwen3 8B	**87.10**	87.03	80.85
DS Llama 8B	84.21	83.20	**84.53**

Table 5: Comparison of pooling strategies for LLaMA-8B and Qwen-8B.

Finally, we evaluate the impact of max pooling. We use three settings: no pooling at all, pooling only during the first compression, and pooling during every compression step. As shown in Figure 5, for DS Llama 8B, the performance across all settings is relatively similar. However, for Qwen3 8B, performing pooling at every compression step leads to a significant drop in performance.

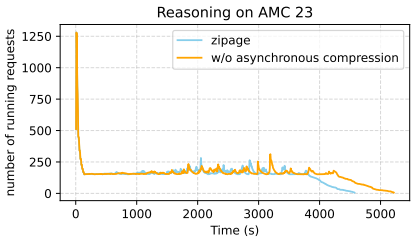
Although pooling does not show significant improvements in this scenario, it has proven to be effective in settings based solely on attention scores. The effectiveness of pooling may stem from its ability to retain some important tokens that are temporarily not attended to by the observation window. The global score serves a similar purpose,

but demonstrates better performance. Introducing pooling alongside the global score may, in fact, hinder the eviction of less important KV cache entries. However, since the first compression step lacks global scores, pooling can still play a useful role. Therefore, we recommend using pooling at first compression.

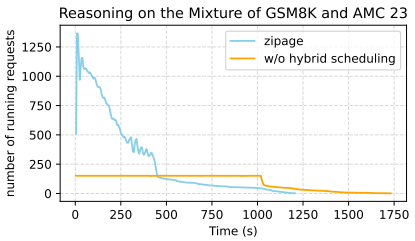
Based on the aforementioned results, we recommend the configuration of $\lambda = 0.2$, $\tau = 0.4$, and $\alpha = 0.8$, with pooling applied only during the first compression step. Although these suggestions are derived from evaluations on a single dataset and therefore have limited generalizability, they have already achieved relatively optimal performance.

D Additional Information of Ablation Experiments

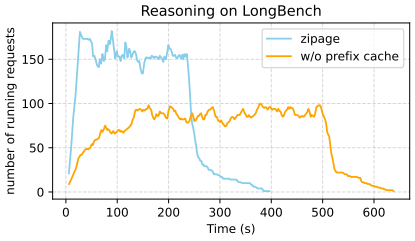
In this section, we provide additional information during the inference process, such as the number of running requests, the number of waiting requests, and block utilization rates.



(a)



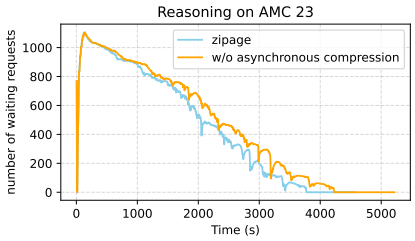
(b)



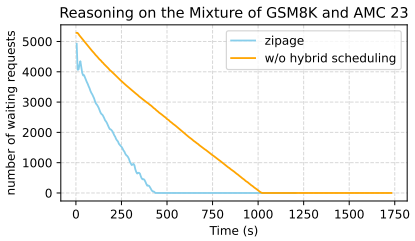
(c)

Figure 19: Number of running requests during inference.

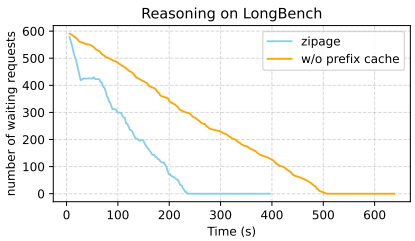
Figure 19 illustrates the number of running requests during inference. As shown in Figure 19 (b), without hybrid scheduling, the number of running requests remains at or below the maximum concurrency. In contrast, with hybrid scheduling enabled, the concurrency starts very high due to a large number of requests requiring only short responses. Figure 19 (c) compares the impact of enabling prefix caching. With prefix caching, the concurrency quickly reaches a high level, whereas without prefix caching, the concurrency increases more gradually.



(a)



(b)

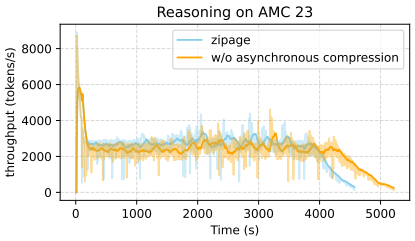


(c)

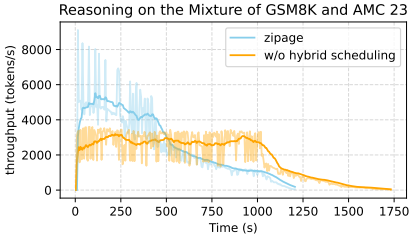
Figure 20: Number of waiting requests during inference.

Figure 20 illustrates the number of waiting requests during inference. When the waiting queue is non-empty, the inference engine operates at full scheduling capacity. In this case, the slope of the waiting request curve indicates the request processing speed. Asynchronous compression, hybrid scheduling, and prefix caching all provide significant acceleration.

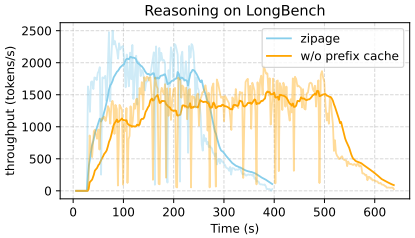
Figure 21 illustrates the real-time throughput during inference. Under request saturation, asynchronous compression, hybrid scheduling, and prefix caching deliver significant throughput improve-



(a)

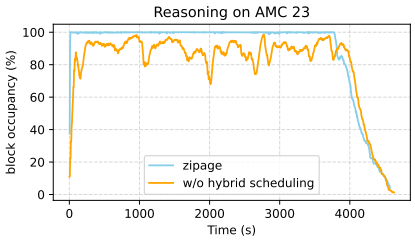


(b)

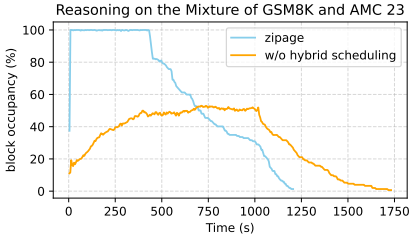


(c)

Figure 21: Throughput during inference.



(a)



(b)

Figure 22: Block occupancy during inference.

ments across various scenarios.

As previously mentioned, constrained scheduling limits concurrency, resulting in some blocks being underutilized. Figure 22 further illustrates the block utilization rate. On the mixed workload, which includes many requests with both short inputs and outputs, this underutilization becomes more pronounced, with less than half of the blocks being utilized most of the time.

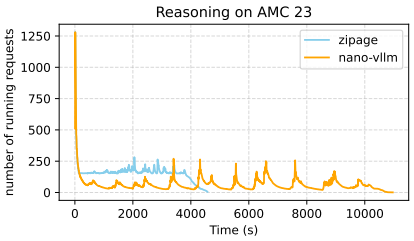
E Additional Information of Comparison with PagedAttention

In this section, we provide additional comparative information between Zipage and Nano-vLLM.

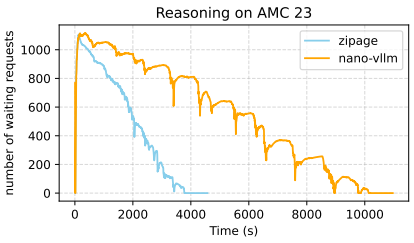
First, Figure 23 illustrates the number of running and waiting requests during inference on AMC23 using the Qwen3 model with different inference engines. The figure shows that the number of running requests in Nano-vLLM exhibits periodic fluctuations, consistent with the throughput variations in Figure 7 (a), while the step decoding time Figure in 7 (b) remains nearly constant. This indicates that the primary factor limiting throughput is concurrency, or more specifically, constrained concurrency caused by limited memory. In contrast to Nano-vLLM, Zipage maintains consistently high throughput throughout because, under the Zipage framework, the maximum memory usage per request is capped at a predefined limit, rather than continuously growing as the sequence length increases.

Figure 24 illustrates the number of running and waiting requests during inference on the mixture of GSM8K and AMC 23 using the Qwen3 model with different inference engines. In this mixed workload, the initial performance of Zipage and Nano-vLLM is very similar, as short-response requests dominate at the beginning. However, as long-response requests occupy a large number of KV cache blocks, requests in Nano-vLLM's waiting queue experience longer delays, significantly increasing total execution time and reducing TPS.

Figure 25 illustrates the number of running and waiting requests during inference on LongBench using the Qwen3 model with different inference engines. We observe that both throughput and the number of running requests are higher with Zipage. However, the total time taken by Zipage to complete inference is slightly longer, likely due to some excessively long outputs. While the average output length on LongBench is only 400 tokens, occasional requests may produce significantly longer outputs, which can substantially impact the total time.



(a)



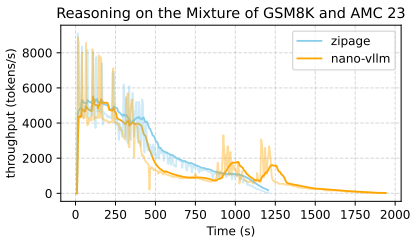
(b)

Figure 23: The number of running and waiting requests during inference during inference on AMC23.

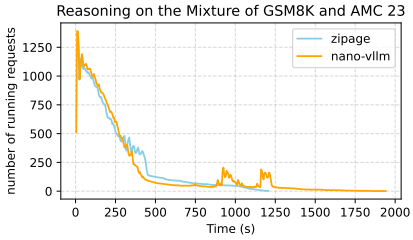
F Experiments on Models of Different Scales.

In this section, we report experimental results for Qwen3 models of different sizes. For all experiments, Zipage uses a budget of 2048. Figure 26 presents the TPS and pass@1 metrics for inference on AMC23 under the Zipage and Nano-vLLM frameworks. Across all model sizes, Zipage achieves significant throughput improvements. Additionally, the performance exceeds 95% of Full KV for all sizes except 14B, which is slightly below 95%.

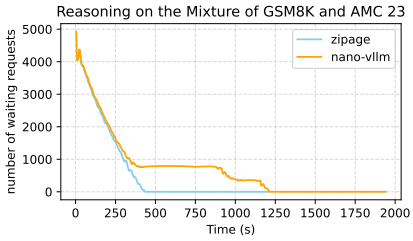
Real-time throughput, average decoding time per step, and the proportion across different concurrency ranges for the 0.6B, 14B, and 32B models are shown in Figures 27, 28, and 29, respectively. Figure 30 provides details for the DS Llama 8B model.



(a)



(b)



(c)

Figure 24: The real-time throughput, number of running and waiting requests during inference during inference on the mixture of GSM8K and AMC 23.

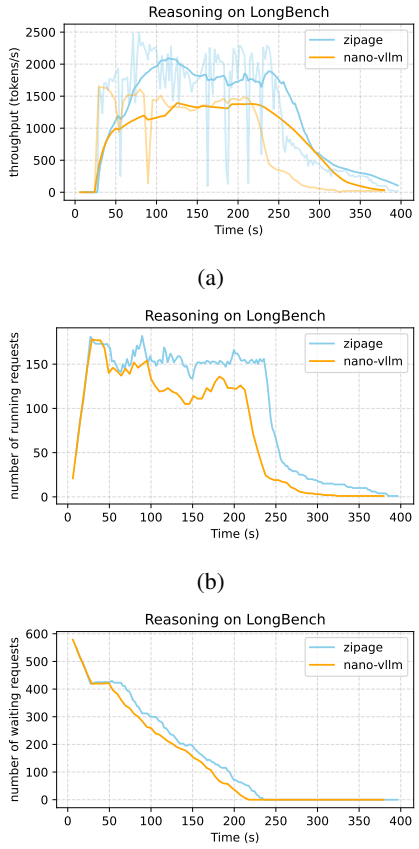


Figure 25: The real-time throughput, number of running and waiting requests during inference during inference on LongBench.

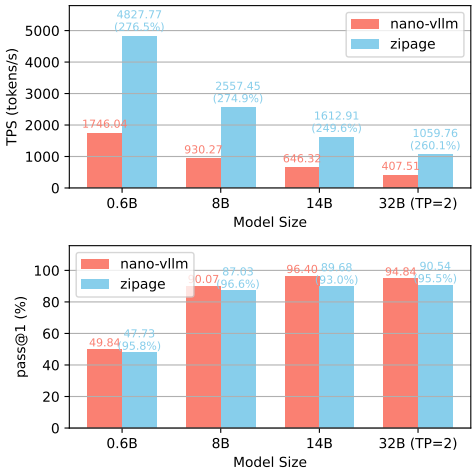


Figure 26: TPS and pass@1 performance across different model sizes. TP=2 indicates that the model is run on two GPUs using tensor parallelism, while all other experiments are conducted on a single GPU by default.

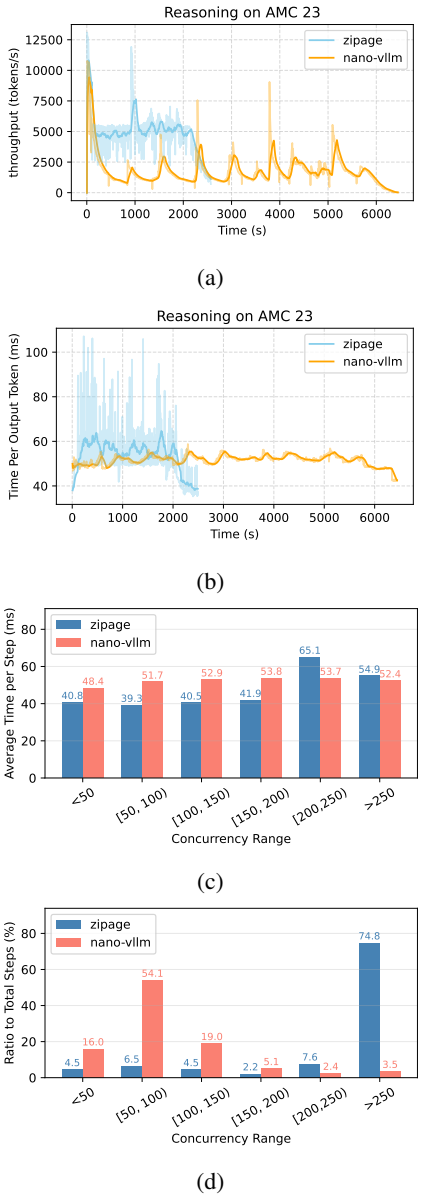
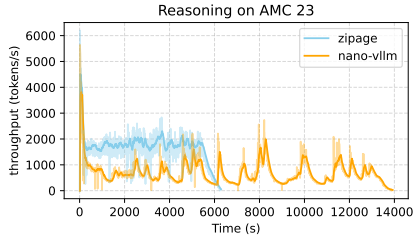
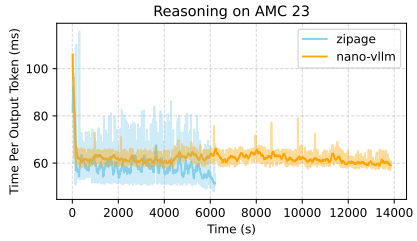


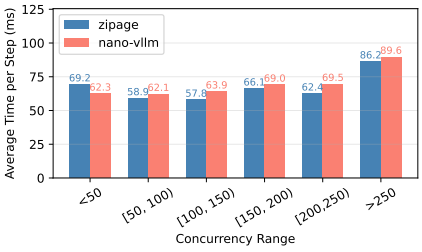
Figure 27: The figure shows Qwen3 0.6B's performance using Zipage or Nano-vLLM on AMC 23, including:(a) real-time throughput, (b) per-step real-time decoding time, (c) average per-step time at different concurrency range, (d) and the ratio of steps to total steps under different concurrency range.



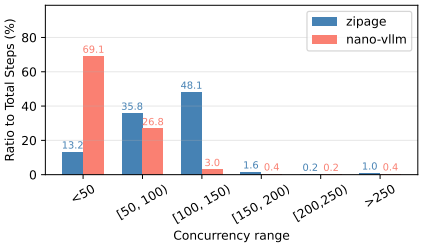
(a)



(b)

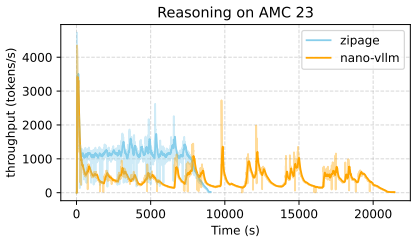


(c)

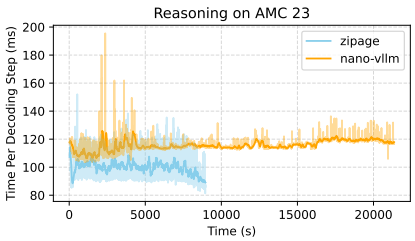


(d)

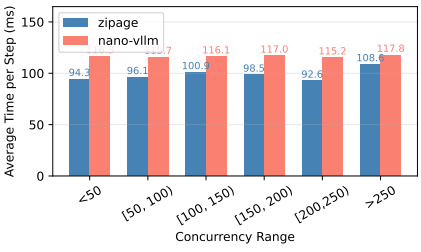
Figure 28: The figure shows Qwen3 0.6B's performance using Zipage or Nano-vLLM on AMC 23, including:(a) real-time throughput, (b) per-step real-time decoding time, (c) average per-step time at different concurrency range, (d) and the ratio of steps to total steps under different concurrency range.



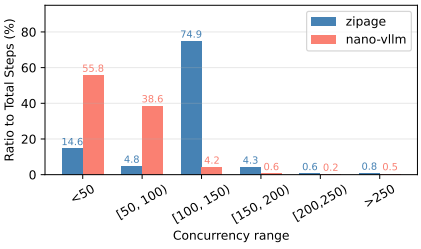
(a)



(b)

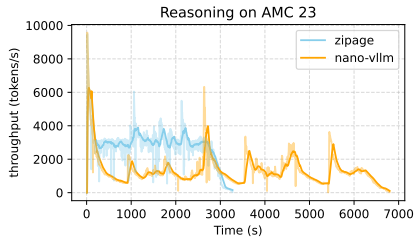


(c)

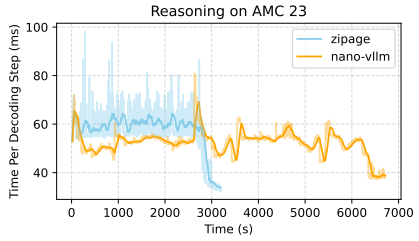


(d)

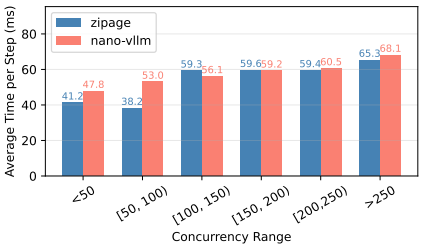
Figure 29: The figure shows Qwen3 32B's performance using Zipage or Nano-vLLM on AMC 23, including:(a) real-time throughput, (b) per-step real-time decoding time, (c) average per-step time at different concurrency range, (d) and the ratio of steps to total steps under different concurrency range. (Tensor parallelism = 2)



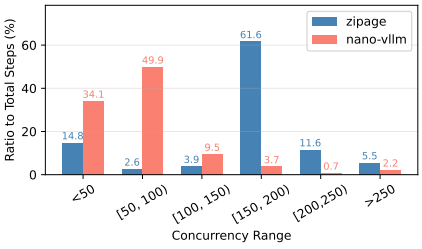
(a)



(b)



(c)



(d)

Figure 30: The figure shows DS Llama 8B's performance using Zipage or Nano-vLLM on AMC 23, including:(a) real-time throughput, (b) per-step real-time decoding time, (c) average per-step time at different concurrency range, (d) and the ratio of steps to total steps under different concurrency range.