AdaptCache: KV Cache Native Storage Hierarchy for Low-Delay and High-Quality Language Model Serving

Abstract

Large language model (LLM) applications often reuse previously processed context, such as chat history and documents, which introduces significant redundant computation. Existing LLM serving systems address such redundant computation by storing the KV caches of processed context and loading the corresponding KV cache when a new request reuses the context. Further, as these LLM applications scale, the total size of KV caches becomes excessively large and requires both DRAM and SSD for full storage.

However, prior work that stores KV caches in DRAM and SSD suffers from high loading delays, as most KV cache hits come from SSD, which is slow to load. To increase the KV cache hit rate on DRAM, we identify lossy KV cache compression as a promising approach. We design a lossy compression system that decides the compression algorithm, compression rate and device placement for each KV cache entry to maximise DRAM hits and minimise loading delay without significantly degrading generation quality. Compared to various static compression baselines across three tasks, our system AdaptCache achieves 1.43–2.4 × delay savings at the same quality and 6–55% quality improvements at the same delay.

1 Introduction

With the accelerating adoption of Large Language Models (LLMs), inference systems must provide low-delay responses to meet growing user demands. To achieve this, many systems store and reuse the KV cache of previously processed contexts (*e.g.,* chat history) to reduce redundant computation and speed up inference. However, due to the rapid growth of LLM applications, the size of KV caches also quickly increases and exceeds the storage capacity of both GPU and CPU memory. For example, even with two H100 nodes (each with 80GB GPU memory) and a total of 150GB CPU memory, models like *Llama-3.1-70B-Instruct* can only store the KV caches of 500K tokens, roughly corresponding to the chat histories of 30 users. This issue becomes more severe in agentic applications, where even a single run involves multiple agents that continuously exchange the text with each other, resulting in a context of millions of tokens and the corresponding KV cache size of 3TB. This underscores the need to fully leverage hierarchical storage within the datacenter to ensure high cache hit rates, reduce delay, and increase throughput.

There is a line of work [2, 3] that explores storing KV caches in hierarchical storage (in this paper we focus on DRAM and SSD). However, we observe that these approaches still have high delay, because high-speed devices have limited storage space, which forces most of the KV caches to be stored to SSD, leading to slow KV cache loading time overall. To allow more cache hits on DRAM, we identify lossy KV cache compression [11, 13] as a promising approach: by lossily compressing the KV caches, we can store much more KV cache entries in CPU memory without significant degredation of generation quality[1].

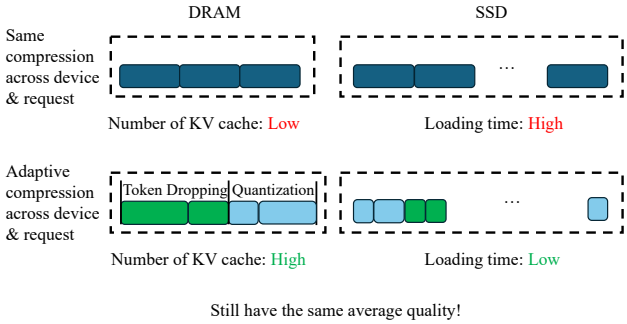

Figure 1: *AdaptCache increases cache hit in high-speed device and reduces loading time in low-speed device while maintaining high generation quality*

As a result, our goal is to design a lossy KV cache storage system on a hierarchical storage (DRAM and SSD) that ensures low inference latency with minimum quality degredation.

This paper argues that, instead of applying the same compression (*i.e.,* using the same compression algorithm and compression rate across devices) to all KV caches, we need to adaptively adjust the compression algorithm, compression rate and device placement to achieve optimal KV cache compression. Concretely:

- Compression algorithm needs to be aware of the context - Some texts are only important at the beginning and the end (which is suitable for token dropping), but some texts aim to provide new information for the LLM (where quantization might be better).
- Compression ratio and device placement need to be aware of the content and the reuse frequency. Some KV caches are frequently reused and easy-to-be-compressed and they need to be stored in CPU memory with a high compression ratio, but some KV caches are not reused frequently and difficult to compress, so it may be suitable to place them on disk with a low compression ratio.

Figure 1 shows that, if we optimally adapt the compression algorithm, rate and device, the potential of improvement is huge: we can achieve much higher KV cache hit rate on CPU and much lower average loading time from disk, while still having comparable accuracy as applying same compression across devices.

To address these challenges, we propose AdaptCache, the first hierarchical KV cache storage system leveraging lossy KV cache compression. The core of AdaptCache is a utility metric that allows us to quantify the effect of storing KV caches in terms of both quality and loading delay. Then, we use *marginal utility gain* to evaluate the effect of each design decisions – compression algorithm, compression rate, cache placement and eviction – and pick the design

[1]Generation quality is defined as the similarity between the generated answer after compression and the original prefill answer, measured by task-specific metrics, *i.e.,* F1, ROUGE-L [9] or CodeBLEU [12].

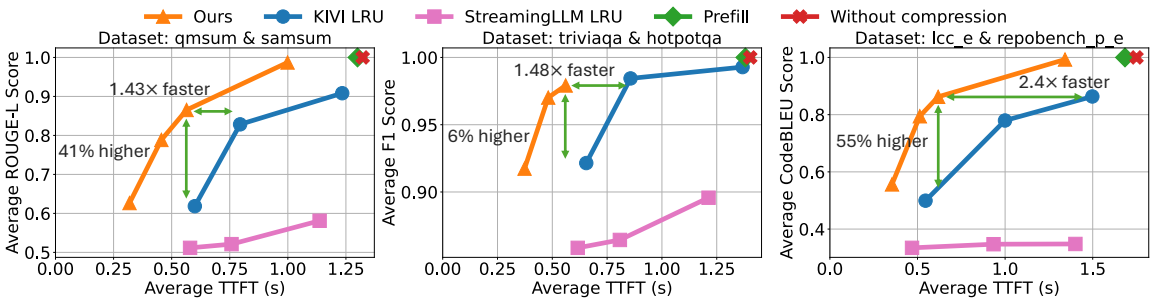

Figure 2: *AdaptCache achieves 1.43–2.4 × lower TTFT and 6–55% higher quality compared to fixed compression method and rate*

decision with the highest marginal utility gain. Acknowledging that this design is greedy and not necessarily optimal, we argue that it suffices to deliver significant delay reduction and accuracy improvement in practice, and optimal solution is not trackable since it is NP-hard.

To evaluate AdaptCache, we use the popular *Llama-3.1-8B-Instruct* model and three LongBench datasets. Compared to the strongest fixed compression baseline, AdaptCache is 1.43–2.4× faster at the same quality and improves quality by 6–55% under the same delay.

2 AdaptCache Design

AdaptCache has three main components: an estimator, a policy optimizer, and an executor. The estimator does offline profiling to estimate the quality–delay curve for each entry, compression method and storage device. The policy optimizer makes compression decisions for incoming and existing KV cache entries. The executor implements the compression decisions.

Estimator: Our offline profiler uses dummy questions to measure the transfer delays of each storage device and decompression overhead. It also samples ten entries from each dataset, using questions generated by GPT 4o, to generate the quality–compression rate curve for each compression method. We estimate the future cache hit frequency of one KV cache entry using its historical hit frequency.

Utility definition: The objective of AdaptCache is to minimize average system delay while maintaining high quality. To capture this trade-off, we compute a quality-delay weighted sum for each KV cache entry. Since some entries are reused more frequently than others, storing these in limited storage can increase the overall hit rate thus reducing delay. Therefore, we define each entry's utility as the product of its estimated future occurrence frequency and its quality-delay weighted sum:

$$Utility(i) = Freq(i) \cdot (\alpha \times Quality(i, M_i, R_i) - \frac{size(i, M_i, R_i)}{Bandwidth}$$

where i represents the i-th KV cache entry, M_i and R_i are the compression method and rate used.

The objective of AdaptCache can be expressed as to maximise the total utility across all KV cache entries. Mathematically, this is an NP-hard Multi-Choice Knapsack Problem (MCKP). We adopt a greedy approach based on the textbook solution [7] that achieves Linear Programming Optimality. When a new KV cache entry is generated, we compute the following metric for both the new entry and all entries in storage. We define

$$\frac{Utility(i, m) - Utility(i, n)}{size(i) \times (m - n)}$$

as the **marginal utility drop**, *i.e.,* the utility drop per unit of space saved if we compress furthermore or evict the entry. m and n are different compression rates.

For each storage tier, we will greedily decide the compression choice (to evict, store in full quality or compress each entry) that results in the minimal marginal utility drop.

3 Preliminary Results

Evaluation setup: We use 1,100 contexts from six LongBench datasets [1, 4–6, 10, 15, 16] to evaluate AdaptCache, covering three task types, summarization, question answering (QA) and coding. These datasets do not contain the timestamps for request arrival, thus we follow previous works to generate request arritval timestamps using Poisson distribution with different request rates [3, 8, 14]. Experiments are run on one NVIDIA A100 GPU (100 GB DRAM, 400 GB SSD) using *Llama-3.1-8B-Instruct* model. The disk reading throughput is 1 GB/s. The baselines we compare are:

- *Without Compression* denotes offloading KV cache to DRAM and SSD without compression.
- *KIVI LRU* / *StreamingLLM LRU* use KIVI [11] and StreamingLLM [13] with fixed compression rates, and offload KV cache to DRAM and SSD using a Least Recently Used (LRU) policy.
- *Prefill* denotes prefilling (recomputation).

Overall results: As shown in Figure 2, across three tasks, AdaptCache consistently reduces time to first token (TTFT) compared to all baselines. Compared to naive prefill and offloading, AdaptCache reduces TTFT by 56%, achieving within 15% quality drop. Compared to KIVI, AdaptCache reduces TTFT by 69% at the same quality. Finally, compared to StreamingLLM, AdaptCache greatly improves quality by 15–89% at the same TTFT.

Understanding AdaptCache's improvements: AdaptCache outperforms approaches that use no compression or fixed compression rates by detecting speed differences between devices and aggressively compressing the KV cache. This strategy significantly increases the hit rate on high-speed devices, leading to much lower loading delay and reduced TTFT. For example, in the coding task, KIVI LRU achieves a DRAM cache hit rate of 38% when quantized to 2 bits. In contrast, our method achieves hit rates of 81%, 56%, 44%, and 11%, for different weights between delay and quality. AdaptCache also prioritises compressing and storing KV cache from longer contexts or those with high information redundancy, and selects optimal compression algorithms for each entry, thereby ensuring overall high response quality.

References

[1] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

[2] Weijian Chen, Shuibing He, Haoyang Qu, Ruidong Zhang, Siling Yang, Ping Chen, Yi Zheng, Baoxing Huai, and Gang Chen. {IMPRESS}: An {Importance-Informed}{Multi-Tier} prefix {KV} storage system for large language model inference. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 187–201, 2025.

[3] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. {Cost-Efficient} large language model serving for multi-turn conversations with {CachedAttention}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 111–126, 2024.

[4] Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. Samsum corpus: A human-annotated dialogue dataset for abstractive summarization. *arXiv preprint arXiv:1911.12237*, 2019.

[5] Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. Longcoder: A long-range pre-trained language model for code completion. In *International Conference on Machine Learning*, pages 12098–12107. PMLR, 2023.

[6] Mandar Joshi, Eunsol Choi, Daniel S. Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension, 2017.

[7] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Multidimensional knapsack problems. In *Knapsack problems*, pages 235–283. Springer, 2004.

[8] 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 *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[9] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[10] Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023.

[11] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024.

[12] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020.

[13] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

[14] Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua, and Colin Raffel. mt5: A massively multilingual pre-trained text-to-text transformer. *arXiv preprint arXiv:2010.11934*, 2020.

[15] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*, 2018.

[16] Ming Zhong, Da Yin, Tao Yu, Ahmad Zaidi, Mutethia Mutuma, Rahul Jha, Ahmed Hassan Awadallah, Asli Celikyilmaz, Yang Liu, Xipeng Qiu, et al. Qmsum: A new benchmark for query-based multi-domain meeting summarization. *arXiv preprint arXiv:2104.05938*, 2021.