Efficient and Accurate KV-cache Management for Long-Sequence LLMs

Yuzhen Mao [1] **Qitong Wang** [2] **Martin Ester** [1] **Ke Li** [1]

Abstract

Key-Value (KV) cache plays a pivotal role in accelerating inference in large language models (LLMs) by storing intermediate attention outputs, thereby avoiding redundant computation during auto-regressive generation. However, the cache's memory footprint scales linearly with sequence length, often resulting in memory bottlenecks on constrained hardware. While prior work has explored offloading KV-cache to the CPU and maintaining a reduced subset on the GPU, these approaches frequently suffer from imprecise token prioritization and degraded performance in long-generation tasks such as multi-turn dialogues and chain-of-thought reasoning. In this paper, we propose a novel KV-cache management strategy that integrates semantic token clustering with PagedAttention, a memory-efficient paging mechanism. By clustering semantically related tokens and organizing them into a hierarchical, dynamically updateable structure, our method improves cache hit rates and memory bandwidth utilization during CPU-GPU transfers. Experimental results show that our approach significantly enhances inference efficiency while preserving generation quality, particularly in long-sequence scenarios. Notably, it achieves higher accuracy even when operating with only half the memory budget, effectively addressing key limitations of existing KV-cache optimization methods.

1. Introduction

Key-Value (KV) cache is a critical component in modern large language models (LLMs) which stores the intermediate attention outputs for each token, allowing the model to reuse these computations in subsequent forward passes. This is particularly important for auto-regressive generation tasks, where tokens are generated one at a time. By caching these values, the model avoids redundant calculations, dramatically reducing the computational cost and time required for generating long sequences. However, the main challenge for a KV cache lies in its memory consumption. As the generated sequence grows longer, the required cache size increases linearly, potentially leading to out-of-memory errors on devices with limited RAM.

Recent research (Zhang et al., 2024b; Tang et al., 2024; Xiao et al., 2023) has revealed that despite the growing size of the KV cache, a small subset of tokens plays a disproportionately important role in maintaining generation accuracy. This insight suggests that we can significantly reduce inference time by selectively loading only these crucial tokens, without compromising the quality of the output. Some research (Chen et al., 2024a; Lee et al., 2024; Chen et al., 2024b) takes this approach further by offloading the KV-cache to the CPU and dynamically maintaining a subset of the most significant KV-cache on the GPU. However, many previous methods do not identify and prioritize these critical parts of the KV-cache in a precise way so that the hit rate of the truly important tokens is low. Moreover, in scenarios involving long-generation tasks, such as long-context summarization, multi-step reasoning, and extended chain-of-thought (CoT) generation, previous methods experience significant performance degradation (Li et al., 2024b).

To address these challenges, we propose an innovative approach, which we call IceCache, that integrates token clustering with a currently prevalent method – PagedAttention (Kwon et al., 2023) which stores the KV-cache in non-contiguous paged memory. As illustrated in Figure 1, by grouping semantically related tokens into pages, our approach aims to enhance the hit rate when selecting critical components and increase the transmission bandwidth during the CPU-GPU offloading for the pages. Additionally, by employing a hierarchical data structure that can be efficiently updated during the decoding phase, we can mitigate the performance degradation commonly observed in long-generation tasks in previous studies. This leads to more effective use of the KV-cache especially in the long-generation setting. Particularly, our method aims to optimize memory usage while preserving the model's performance by focusing on the following key aspects:

[1]School of Computing Science, Simon Fraser University, Canada [2]Harvard University, Cambridge, USA. Correspondence to: Yuzhen Mao <yuzhenm@sfu.ca>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

1. Token Clustering for Efficient Storage: In the Prefill stage, instead of storing the KVs sequentially in their original order, we first cluster the tokens based on their similarity in a transformed key-embedding space using a maintainable tree-structured index, called DCI-tree. Tokens belonging to the same cluster are then stored together in the same memory page(s).

2. Query-aware Critical Page Selection: In the Decoding stage, given a specific query, only a subset of pages for each head are loaded to GPU to perform the attention computation for each layer and attention head. These pages are selected based on the presence of critical tokens, which is decided by an Approximated Nearest Neighbour (ANN) algorithm called Multi-level DCI (M-DCI).

3. Efficient Pipelining with CPU-GPU overlapping: IceCache performs M-DCI-based page selection on the CPU in parallel with GPU operations such as attention computation and feedforward layer execution. This pipelined design effectively overlaps computations, significantly hiding the latency introduced by page selection.

We evaluated IceCache under constrained GPU memory budgets on the LongBench (Bai et al., 2023) benchmark using Llama3.1-8B-Instruct. Across diverse tasks, including open-domain QA, multi-hop reasoning, academic reading comprehension, and long-context summarization, IceCache consistently outperformed six state-of-the-art KV-cache baselines in accuracy while maintaining comparable low latency. Notably, IceCache sustained near-oracle performance on most tasks using only a small fraction of the original KV cache size. What's more, IceCache achieves higher accuracy even when operating with only half the memory budget.

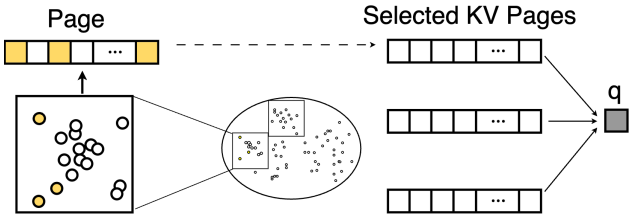


Figure 1: Illustration of the clustering and query-aware page selection mechanism in IceCache. The diagram depicts how tokens (represented as dots) are first grouped into clusters based on their semantic similarity in a transformed key-embedding space. During the decoding stage, given a query q, the zoomed-in section highlights that critical tokens (highlighted in yellow) are grouped in the same cluster and stored together within the same memory page. After performing a query-aware critical page selection, IceCache conducts the approximated attention with the selected KVs and q.

2. IceCache

We propose an innovative approach, named IceCache, that integrates token clustering with KV-cache storage. Our method consists of three steps: (1) Indexing; (2) Page Selection; and (3) Bulk Back-loading. The Indexing step occurs either during the prompt processing phase—when IceCache constructs a hierarchical tree structure, referred to as the DCI-tree, for the prompt key embeddings; or when new window pages are offloaded to the CPU. In this step, similar tokens are grouped into units called nodes, rather than being stored sequentially in virtual memory pages as in PagedAttention. Here, a node denotes a group of data points that share the same parent in the tree hierarchy. The next two steps take place during the token generation (decoding) phase. In the Page Selection step, IceCache employs a fast Approximate Nearest Neighbor (ANN) search algorithm, P-DCI, to independently select the top-k most relevant key pages for each attention head. Finally, in the Bulk Back-loading step, the selected pages are efficiently transferred from the CPU back to the GPU. IceCache overlaps the DCI search (a CPU-intensive operation) with ongoing GPU computations, thereby minimizing additional latency.

We provide further details on each of these three steps in the following subsections and illustrate the method in Fig 2.

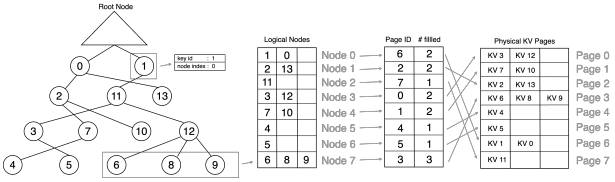


Figure 2: Illustration of DCI-tree and IceCache: The hierarchical structure on the left visualizes the result of indexing key embeddings, DCI-tree, where each tree node stores metadata for the tokens such as the key ID and node index. The tables on the right depict the mapping between nodes in the DCI-tree and the corresponding pages in physical memory. For each selected node, a mapping table is used to locate the memory region containing the associated key-value embeddings.

2.1. Indexing: Clustering Key Embeddings into a Hierarchical Tree

PagedAttention (Kwon et al., 2023) is a memory management strategy designed to optimize attention computation in LLMs by organizing key-value pairs into sequential memory pages. It stores these key-value pairs based on their original token indices, ensuring that tokens appearing consecutively in the input sequence are placed contiguously in memory. This organization minimizes memory fragmentation, allowing for more efficient memory access during decoding and ultimately improving computational throughput. To inherit the benefits of PagedAttention, several subsequent

KV-cache optimization techniques, such as Quest (Tang et al., 2024) and ArkVale (Chen et al., 2024a), have been developed based on its principles. They focus on estimating the importance of each page during KV-cache selection to approximate attention computation more efficiently.

IceCache also organizes key-value embeddings into pages, but takes a fundamentally different approach during its Indexing stage. Instead of relying on the token's original order, IceCache constructs a hierarchical tree structure for each attention head, called a DCI-tree, which clusters tokens based on the semantic similarity of their key embeddings. Each node in the DCI-tree represents a small group of semantically related tokens that share a common parent, effectively forming a localized cluster. From a memory system perspective, IceCache maps each of these nodes directly to a memory page, thereby preserving semantic locality in storage and enabling efficient access during decoding.

By clustering semantically similar tokens into the same nodes/pages, IceCache facilitates more targeted and efficient retrieval during decoding. This page structure allows the model to load cached key-value embeddings at the granularity of tree nodes, improving reuse and reducing memory access overhead. As new windows of tokens (e.g., from a sliding window during long-context processing) are offloaded to the CPU, IceCache incrementally inserts each token into the appropriate node in the DCI-tree based on its key embedding. If a node exceeds the maximum page size, new pages are dynamically allocated. This adaptive tree maintenance ensures that the index remains efficient and semantically coherent even as the context evolves, making it particularly effective for long-sequence generation.

2.2. Page Selection: Head-specific ANN Search with Fine-grained Retrieval

During the decoding phase, given a query, IceCache performs a head-specific page selection to identify the most relevant key pages for each attention head. Leveraging the hierarchical DCI-tree built during indexing, we apply a fast ANN search method mentioned in Section B.3, M-DCI, to find the top-k pages that are closest to the current query embedding for each head independently.

This design contrasts sharply with prior methods like Quest and PQCache (Zhang et al., 2024a), which either retrieve all pages indiscriminately or use a coarse global selection strategy shared across heads. In contrast, IceCache's head-specific search recognizes that different heads often attend to different semantic aspects of the input, and thus benefit from customized retrieval strategies. This per-head granularity leads to improved attention relevance and model accuracy.

Furthermore, to reduce latency, the M-DCI is executed on the CPU in parallel with ongoing GPU computations. While the GPU processes the next tokens in the decoding pipeline, M-DCI searches for the best-matching pages, which are then bulk-loaded into GPU memory just-in-time. This overlapping of CPU-GPU workloads helps hide the cost of ANN search and data transfer, contributing to low inference latency even in long-sequence settings. We explain its details in the following section.

2.3. Efficient Pipelining with Bulk Back-loading

In this section, we present how to optimize the IceCache workflow to bulk back-load the selected pages using two CPU and GPU backload buffers. In addition, we carefully design an efficient pipeline of DCI querying, page back-loading, and next-token decoding. By overlapping DCI query with back-loading and decoding, IceCache effectively hides the query latency and achieves fully efficient decoding.

We illustrate our bulk back-loading workflow in Figure 4. After identifying the most relevant pages (indicated by color), we filter out those already resident in GPU memory from previous token generations. The remaining pages are then aggregated into a pre-allocated CPU-memory buffer, enabling a single high-throughput PCIe transfer into a pre-allocated GPU-memory buffer. Once the pages arrive in GPU memory, we scatter them directly into their corresponding entries in the KV-Cache table.

Figure 5(b) illustrates the IceCache pipelining (other minor stages are omitted for simplicity). Based on observations that the hidden states of consecutive LLM layers are highly similar (Liu et al., 2024; Li et al., 2024a), we decouple DCI querying from page back-loading and next-token decoding by speculatively querying the DCI indexes using the previous layer's hidden states. Specifically, while back-loading pages and decoding tokens for layer i, we simultaneously issue a DCI query over layer i's hidden states to identify the critical pages for layer i+1. This overlapping fully hides the query latency in IceCache.

3. Experiments

3.1. Settings

We apply our method to Llama3.1-8B, one of the most popular open-source LLMs employing group-query attention (GQA) (Ainslie et al., 2023). We first evaluate the recall in retrieving important tokens, followed by performance testing on six benchmarks from LongBench (Bai et al., 2023). For baseline comparisons, we select state-of-the-art KV cache optimization methods, including StreamingLLM (Xiao et al., 2023), H2O (Zhang et al., 2024b), ArkVale (Chen et al., 2024a), SnapKV (Li et al., 2024a), Quest (Tang et al., 2024), SparQ (Ribar et al., 2023), RocketKV (Behnam et al., 2025) and PQCache (Zhang et al., 2024a) as baselines. As prior research indicates, the initial

layers of the model exhibit relatively low sparsity. Therefore, neither IceCache nor baseline methods are applied to the first two layers of the models. For IceCache, we report results for two variants: (1) **IceCache**: without prefetching; (2) **IceCache (P)**: with prefetching. Specifically, at layer n, the query from this layer prefetches the DCI database of layer $n+1$, and utilizes the search results from layer $n-1$ for attention computation. In this setting, attention computations can proceed without waiting for query completion, thus hiding the latency.

3.2. Estimation Accuracy

We start by evaluating the accuracy of the token clustering and important-pages selection outlined in Section 2.2. Using data from LongBench(Bai et al., 2023), we collect both the actual token rankings and the estimated rankings based on the tokens DCI returns. Recall accuracy is defined for varying values of k as the proportion of overlap between the DCI-selected top-k token set (DCI_k) and the true top-k token set (True_k): Recall $= |\text{DCI}_k \cap \text{True}_k| / |\text{True}_k|$.

Figure 3 presents the recall accuracy of the approximate top-k results over 10 decoding iterations, across different k values ($k = \{1, 3, 5\}$) for three different layers (8, 18, 28). As shown in the figure, IceCache ensures at least 90% accuracy for top-1 recall and consistently achieves over 80% accuracy for other k values.

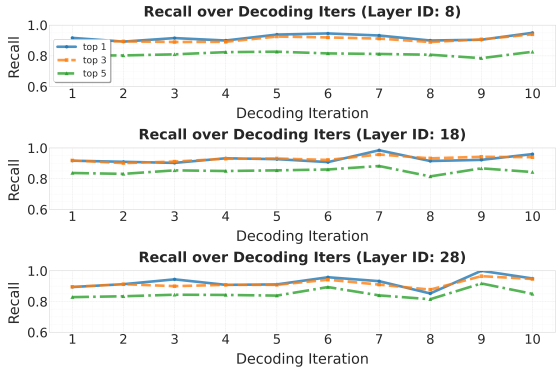


Figure 3: Recall accuracy (the proportion of tokens predicted to be among the top-k that indeed belong to the top-k) of $k = \{1, 3, 5\}$ over 10 decoding iterations for layers 8, 18, 28.

3.3. LongBench Performance

In this experiment, all methods are evaluated under a constrained memory budget of 256 or 128 tokens, except for Full-KV, which serves as the upper-bound baseline with unrestricted memory. We assess performance across six tasks from the **LongBench** benchmark suite. These tasks collectively assess various aspects of long-context understanding, including question-and-answering, summarization, and information retrieval.

Table 1 presents results on Llama3.1-8B-Instruct. From the table, IceCache maintains its superiority in the low-memory setting, offering performance nearly indistinguishable from the Full-KV baseline. In the *GovReport* task, IceCache reaches 34.6 and 33.4 (with and without prefetching, respectively), outperforming all other memory-efficient baselines, including RocketKV (26.7) and PQCache (27.2). These results underscore IceCache's ability to recall and utilize essential long-range context, which is crucial for summarization-heavy tasks. Furthermore, IceCache (P) achieves the best or second-best results in *Passage Retrieval*, *TriviaQA*, *NarrativeQA*, indicating that the prefetching approximation delivers reliable performance across both open-domain QA and structured summarization benchmarks.

We also evaluate the performance of IceCache using an even smaller memory budget of 128 tokens, pushing the limits of efficient decoding. We observe that while reducing the memory budget from 256 to 128 slightly impacts performance, IceCache remains highly competitive and often outperforms existing baselines that use twice the budget. Specifically, IceCache with a budget of 128 achieves performance very close to the 256-budget version across multiple tasks. In some cases, such as gov-report and qasper, it even achieves the best or second-best results. These results highlight the effectiveness of IceCache 's token clustering and page selection strategy, enabling substantial memory savings without sacrificing accuracy.

Table 1: LongBench evaluation for Llama3.1-8B-Instruct. IceCache shows strong performance even under half of the memory budget (128). We bold the best score and underline the second-best score for each task.

Budget	Method	gov-report	narrativeqa	qasper	hotpotqa	triviaqa	pass-ret.
N/A	Full-KV	35.2	30.2	45.5	55.5	91.7	99.5
256	Exact-TopK	34.8	30.7	44.7	55.0	92.2	99.5
256	SnapKV	N/A	0.0	0.0	47.5	75.0	99.0
256	Quest	9.8	3.0	22.4	14.9	17.3	7.5
256	SparQ	13.7	5.0	38.0	19.7	32.1	28.8
256	RocketKV	26.7	30.3	43.5	54.9	91.0	99.5
256	PQCache	27.2	28.7	43.3	**55.2**	91.1	99.0
256	ArkVale	22.2	23.9	9.0	14.4	90.0	91.1
256	IceCache	**34.6**	**30.6**	**44.7**	**55.2**	**92.0**	**100.0**
256	IceCache (P)	33.4	30.4	44.5	54.5	91.7	99.6
128	IceCache	33.5	30.0	**44.7**	55.0	91.3	**100.0**
128	IceCache (P)	32.3	29.3	43.5	54.1	90.8	99.5

4. Conclusion

This paper introduces a novel hierarchical database, the DCI-tree, enabling lightweight updates and dynamic token management for efficient KV-cache handling. We further propose IceCache, an end-to-end page-based KV-cache manager with efficient GPU-CPU offloading and recall. Extensive experiments across diverse long-context tasks demonstrate IceCache's efficacy. It achieves state-of-the-art performance by significantly reducing decoding memory usage and improving time per output token under constrained KV-cache budgets (128 tokens), with negligible accuracy loss.

References

Ainslie, J., Lee-Thorp, J., De Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

Bai, Y., Lv, X., Zhang, J., Lyu, H., Tang, J., Huang, Z., Du, Z., Liu, X., Zeng, A., Hou, L., et al. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

Behnam, P., Fu, Y., Zhao, R., Tsai, P.-A., Yu, Z., and Tumanov, A. Rocketkv: Accelerating long-context llm inference via two-stage kv cache compression. *arXiv preprint arXiv:2502.14051*, 2025.

Chen, R., Wang, Z., Cao, B., Wu, T., Zheng, S., Li, X., Wei, X., Yan, S., Li, M., and Liang, Y. Arkvale: Efficient generative llm inference with recallable key-value eviction. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024a.

Chen, Z., Sadhukhan, R., Ye, Z., Zhou, Y., Zhang, J., Nolte, N., Tian, Y., Douze, M., Bottou, L., Jia, Z., et al. Magicpig: Lsh sampling for efficient llm generation. *arXiv preprint arXiv:2410.16179*, 2024b.

Gupta, A., Dar, G., Goodman, S., Ciprut, D., and Berant, J. Memory-efficient transformers via top-k attention. *arXiv preprint arXiv:2106.06899*, 2021.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Lee, W., Lee, J., Seo, J., and Sim, J. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 155–172, 2024.

Li, K. and Malik, J. Fast k-nearest neighbour search via prioritized dci. In *International conference on machine learning*, pp. 2081–2090. PMLR, 2017.

Li, Y., Huang, Y., Yang, B., Venkitesh, B., Locatelli, A., Ye, H., Cai, T., Lewis, P., and Chen, D. Snapkv: Llm knows what you are looking for before generation. *arXiv preprint arXiv:2404.14469*, 2024a.

Li, Y., Jiang, H., Wu, Q., Luo, X., Ahn, S., Zhang, C., Abdi, A. H., Li, D., Gao, J., Yang, Y., et al. Scbench: A kv cache-centric analysis of long-context methods. *arXiv preprint arXiv:2412.10319*, 2024b.

Liu, Z., Desai, A., Liao, F., Wang, W., Xie, V., Xu, Z., Kyrillidis, A., and Shrivastava, A. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems*, 36, 2024.

Mao, Y., Ester, M., and Li, K. Iceformer: Accelerated inference with long-sequence transformers on CPUs. In *The Twelfth International Conference on Learning Representations*, 2024.

Mohtashami, A. and Jaggi, M. Landmark attention: Random-access infinite context length for transformers. *arXiv preprint arXiv:2305.16300*, 2023.

Nikita, K., Lukasz, K., Anselm, L., et al. Reformer: The efficient transformer. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2020.

Ribar, L., Chelombiev, I., Hudlass-Galley, L., Blake, C., Luschi, C., and Orr, D. Sparq attention: Bandwidth-efficient llm inference. *arXiv preprint arXiv:2312.04985*, 2023.

Tang, J., Zhao, Y., Zhu, K., Xiao, G., Kasikci, B., and Han, S. Quest: Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774*, 2024.

Xiao, G., Tian, Y., Chen, B., Han, S., and Lewis, M. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

Xiao, G., Tang, J., Zuo, J., Guo, J., Yang, S., Tang, H., Fu, Y., and Han, S. Duoattention: Efficient long-context llm inference with retrieval and streaming heads. *arXiv preprint arXiv:2410.10819*, 2024.

Zhang, H., Ji, X., Chen, Y., Fu, F., Miao, X., Nie, X., Chen, W., and Cui, B. Pqcache: Product quantization-based kvcache for long context llm inference. *arXiv preprint arXiv:2407.12820*, 2024a.

Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C., et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36, 2024b.

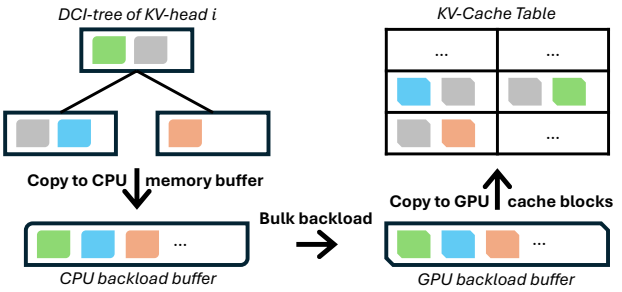


Figure 4: IceCache groups and backloads the selected pages to maximize the PCIe throughput.

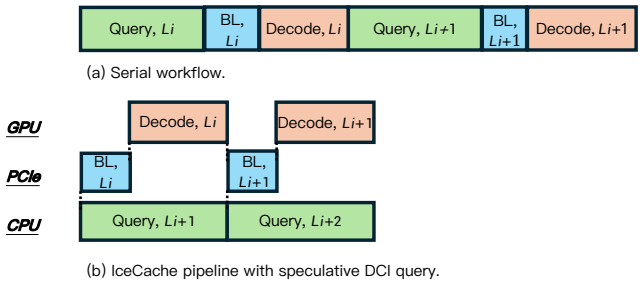


Figure 5: IceCache pipelines and hides the DCI query completely.

A. Related work

Lots of recent methods have aimed to enhance the efficiency of attention mechanisms in large language models, especially for handling long-context inputs. H2O (Zhang et al., 2024b) only keeps a subset of tokens selected by the attention scores to save the memory for KV-cache. StreamingLLM (Xiao et al., 2023) utilizes the initial tokens which they called sink tokens and the most recent tokens to accelerate the attention computation. Methods such as SparQ (Ribar et al., 2023) applies approximate attention by only selecting important indices across the head dimension. Similarly, RocketKV (Behnam et al., 2025) integrates structured caching with selective token retention. DuoAttention (Xiao et al., 2024) reduces memory overhead by splitting attention computations into local and global phases. SnapKV (Li et al., 2024a) uses the last portion of the prompt to select the important key embeddings for the following decoding. PQCache (Zhang et al., 2024a) employs product quantization to manage KV-cache and approximate the attention computation.

PagedAttention (Kwon et al., 2023) is an innovative memory management technique designed to optimize the KV-cache of LLMs. It addresses the challenges by introducing a paging mechanism similar to virtual memory systems in operating systems. This approach divides the KV cache into fixed-size pages, allowing for more efficient memory allocation and management. By doing so, PagedAttention enables better utilization of GPU memory, reducing fragmentation and allowing for longer context windows without sacrificing performance.

Quest (Tang et al., 2024) and ArkVale (Chen et al., 2024a) are two query-aware criticality estimation algorithms built on the PagedAttention. They effectively identify critical KV-cache tokens and perform self-attention selectively on the chosen tokens. For each page, Quest and ArkVale calculate an upper bound using the feature values of the Key vector for each page's criticality estimation. Given all criticality scores of the pages, Top-K pages are chosen to perform approximate self-attention, where K is a preset constant (e.g. 128, 256). Additionally, ArkVale integrates the GPU-CPU offloading into the system to further save GPU memory. However, the main issue with both Quest and ArkVale is that they make all tokens in the query head attend to the same key/value blocks activated by sparse attention, which is too coarse-grained, as the information each token needs to attend to can vary significantly. Instead, IceCache allows each query head to attend to different key/value blocks, which makes the attention-approximation more accurate.

B. Background

B.1. Attention Mechanism

Mathematically, the attention operation takes three matrices as input, $\mathbf{K} \in \mathbb{R}^{m \times d}, \mathbf{Q} \in \mathbb{R}^{n \times d}, \mathbf{V} \in \mathbb{R}^{m \times d'}$, which denote keys, queries and values respectively. Optionally, it may also take in a mask as input, $\mathbf{S} \in \mathbb{R}^{n \times m}$, whose entries are either 0 or 1. The ith rows of $\mathbf{K}$, $\mathbf{Q}$ and $\mathbf{V}$, denoted as $\mathbf{k}_i$, $\mathbf{q}_i$ and $\mathbf{v}_i$, represent the ith key, query, value and output respectively. The entry of $\mathbf{S}$ in the ith row and jth column, denoted as $s_{i,j}$, represents whether the ith query is allowed to attend to the jth key — if it is 1, it would be allowed; if it is 0, it would not be. A common masking scheme is the causal mask, where $s_{i,j}$ is 1 if $i \geq j$ and 0 otherwise. Keys and queries have the same dimension d, and each key is associated with a value, and so the number of keys and values is the same and denoted as m. The attention operation computes the attention weight matrix

$\mathbf{A} \in \mathbb{R}^{n \times m}$. Its entry in the ith row and jth column, denoted as $a_{i,j}$, is computed with the following formula:

$$a_{i,j} = (s_{i,j} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d}}\right)) / \sum_{j'=1}^{m} s_{i,j'} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_{j'}}{\sqrt{d}}\right) \tag{1}$$

The attention matrix $\mathbf{A}$ is typically sparse (Nikita et al., 2020; Gupta et al., 2021), i.e., in each row of $\mathbf{A}$, only a few attention weights have significant (large) values, while the majority of the remaining values are close to zero. Suppose we can somehow identify the k unmasked keys that receive the highest attention weights for each query $\mathbf{q}_i$ without computing the attention weights for all keys. Then, the original attention matrix $\mathbf{A}$ can be approximated by only computing the inner product for the identified keys, which can save significant amount of time and computational resource.

B.2. Generative Inference of LLM

The generative inference process of LLMs primarily comprises two key stages: the prefill (or prompt) stage and the decoding (or generation) stage.

In the prefill stage, the model takes an input prompt sequence of length s_{in} and processes it through all layers of the LLM. During this process, the keys and values for each token in the sequence are computed and stored as part of the KV cache. The decoding stage begins once the prompt has been processed. Here, the model generates output tokens one step at a time, using and updating the KV cache iteratively. For each decoding step, the current token's computation depends on the stored keys and values from previous tokens, allowing the model to maintain context over the sequence. The KV cache thus plays a crucial role in enabling efficient autoregressive generation by reducing redundant computations and maintaining information about past tokens.

B.3. Multi-level DCI

Prioritized Dynamic Continuous Indexing (P-DCI) (Li & Malik, 2017) is an exact, randomized algorithm designed to perform efficient k-nearest neighbour (k-NN) searches in high-dimensional spaces. Unlike traditional methods that rely on space partitioning, P-DCI avoids this by constructing multiple indices, each imposing an ordering of all data points based on their projections onto random vectors. During querying, P-DCI maintains a priority queue to process points in an order that is likely to find nearer neighbours sooner. It computes a dynamic lower bound on the distance to the nearest neighbour, allowing early termination of the search when the bound exceeds the distance to the current best candidates. This approach significantly reduces the number of distance evaluations and memory usage compared to methods like Locality-Sensitive Hashing (LSH).

Multi-level Dynamic Continuous Indexing (M-DCI) (Mao et al., 2024) extends P-DCI by introducing a hierarchical structure to further enhance search efficiency. The index is organized into multiple levels, where each level contains a subset of data points. Points are randomly promoted to higher levels, forming a pyramid-like structure. Each point at a lower level is assigned a parent in the next higher level, typically the nearest neighbour among the promoted points. This creates "cells" or clusters of points sharing the same parent. When querying, the algorithm starts at the top level, using P-DCI to find the k-closest points to the query. It then recursively searches within the cells associated with these points at the next lower level, continuing this process down the hierarchy. This multi-level approach allows M-DCI to focus computational resources on the most promising regions of the index, effectively narrowing down the search space and improving query times, especially in indexes with high intrinsic dimensionality.

C. Pseudocode for IceCache

We separate all tokens into three groups: *sink tokens*, which are the tokens at the very beginning of the input sequence; *window tokens*, which are the most recent tokens; and all the remaining tokens in between. The pages that store sink tokens are referred to as *sink pages*, and those that store window tokens are referred to as *window pages*. We always keep all the sink and window pages in GPU.

We provide the pseudocode below for IceCache. It operates in two main phases: (1) Prefill Phase: During the initial processing of prompt tokens, IceCache allocates paged KV memory per layer and performs self-attention computations. From the third layer onward, it copies KV embeddings to CPU and builds a dynamic index – DCI-tree. This tree enables efficient future lookup of important tokens based on query embeddings. (2) Decode Phase: During autoregressive decoding,

each new token's query embedding is used to retrieve the most relevant KV pages via DCI-based query. Selected pages are back-loaded to GPU on demand, while unimportant pages are offloaded to CPU storage. When a new window page is offloaded, the DCI-tree is incrementally updated to store tokens in this page. The detailed steps are outlined in Algorithm 1. We will explain the mechanisms behind indexing and page selection in the following sections.

Algorithm 1 IceCache

Input: Sequence of tokens $x_{1:I}$, Transformer with L layers, Page size s
Phase 1: Prefill
for $\ell = 0$ to $L-1$ **do**
  Allocate pages and arrange KVs to the pages for layer ℓ
  if $\ell \geq 2$ **then**
    Copy KVs of tokens between sink tokens and window tokens from GPU to CPU (denoted as S_k and S_v)
  end if
  Compute the output from the current self-attention layer ℓ
  if $\ell \geq 2$ **then**
    $T_l \leftarrow$ DCI-INDEXING(S_k, S_v)
  end if
end for
Phase 2: Decode (repeated over time steps $i > I$)
while receive new token x_i with $\mathbf{q}_i$ as its query embedding **do**
  for $\ell = 0$ to $L-1$ **do**
    if Number of tokens in the last page $\geq s-1$ **then**
      Offload the oldest window page p_w from GPU to CPU
      Set Flag to True
    end if
    Append KVs of x_i to the end of the newest window page
    if $\ell \geq 2$ **then**
      $S_l \leftarrow$ PAGE-SELECT$(\mathbf{q}_i, T_l, k)$
      Recall selected pages S_l from CPU to GPU
    end if
    Compute the output from the current self-attention layer ℓ
    if $\ell \geq 2$ **and** Flag is True **then**
      Insert the offloaded p_w to T_l
    end if
  end for
end while

D. Method Details

D.1. Indexing

For each attention head, given a set of pre-computed key embeddings, IceCache first indexes them using a hierarchical tree structure which is obtained by a novel approach called Multi-level DCI (M-DCI). It works by constructing a dynamic index called DCI-tree and applies Prioritized DCI (P-DCI) (Li & Malik, 2017) to each level of the tree recursively (more details are in Section B.3). The data points processed in M-DCI are transformed key embeddings and query embeddings using the following transformation formulas, which we denote as $T_K : \mathbb{R}^d \to \mathbb{R}^{d+1}$ and $T_Q : \mathbb{R}^d \to \mathbb{R}^{d+1}$:

$$T_K(\mathbf{k}_j) = \left[\mathbf{k}_j/c \quad \sqrt{1 - \|\mathbf{k}_j\|_2^2/c^2}\right]^\top \tag{2}$$

$$T_Q(\mathbf{q}_i) = \left[\mathbf{q}_i/\|\mathbf{q}_i\|_2 \quad 0\right]^\top \tag{3}$$

where $c \geq \max_{j'} \|\mathbf{k}_{j'}\|_2$ is at least the maximum norm across all keys. We use the Euclidean distance as the distance function.

At the very beginning of the indexing, all data points are initially placed at the bottom level of the DCI-tree. Subsequently,

some points are randomly selected to be promoted to the next higher level based on a promotion ratio $r < 1$. The ratio r is predefined during DCI-tree initialization and remains fixed throughout the process. After the indexing, we can get the total number of levels in the DCI-tree, denoted as L. The details are presented in Algorithm 2.

Specifically, let n_ℓ denote the number of data points at level ℓ, with level indices starting from the top (i.e., the highest level is $\ell = 0$). Ideally, the number of points satisfies the recurrence relation $n_\ell = r \cdot n_{\ell+1}$. In other words, the distribution over level indices follows a geometric distribution. The probability that a point is assigned to the highest level ($\ell = 0$) is r^{L-1}, while the probability of being assigned to level ℓ (for $1 \leq \ell \leq L-1$) is $r^{L-1-\ell} - r^{L-\ell}$.

After level assignment, each data point at level ℓ is linked to a parent at level $\ell - 1$, defined as the closest point in terms of key embedding distance. This parent assignment is formulated as a 1-nearest neighbor search and is efficiently solved using M-DCI.

In the decoding stage, when a new token is generated, its key embedding is inserted into the appropriate position in the DCI-tree. A level ℓ is first assigned to the new key according to the same random promotion process. Then, its parent at level $\ell - 1$ is determined, and the key is added to the physical memory page corresponding to the node into which it is inserted.

Algorithm 2 DCI-INDEXING

Input: A list S_k of n keys $\mathbf{k}_1, \ldots, \mathbf{k}_n \in \mathbb{R}^d$, A list S_v of n values $\mathbf{v}_1, \ldots, \mathbf{v}_n \in \mathbb{R}^{d'}$
$\{l_1, \ldots, l_n\} \leftarrow$ assign target levels to keys in S_k
Initialize T with empty root node
for $i = 1$ **to** n **do**
    $\{p_i\} \leftarrow \text{QUERY}(\mathbf{k}_i, T, l_i, 1)$
    Insert $\mathbf{k}_i$ into T with parent p_i
end for
Return: T

D.2. Page Selection

As aforementioned, IceCache aims to accelerate self-attention by loading only a limited number of pages into GPU memory for computation. Therefore, the objective of page selection is to maximize the *recall* (or hit rate) of significant keys for a given query. By clustering semantically similar tokens into the same nodes/pages, IceCache enables more targeted and efficient retrieval during decoding. In contrast, methods like Quest (Tang et al., 2024), Arkvale (Chen et al., 2024a), or PQCache (Zhang et al., 2024a) construct pages based on the original token order, which often causes tokens relevant to a given query to be scattered across multiple pages. Retrieving them requires loading entire pages filled with many irrelevant tokens, resulting in unnecessary memory overhead. IceCache mitigates this inefficiency by grouping similar tokens, so relevant tokens tend to be concentrated within fewer pages. As a result, the hit rate of significant keys during decoding increases. The detailed procedure is shown in Algorithms 3 and 4.

Specifically, when computing the attention matrix, given a query vector q_i, we follow the query process described in Section B.3 to identify the top-k keys that are most likely to yield the highest dot-product values with q_i. Once these top-k keys are identified, we load only the pages that contain them. Suppose p pages are loaded, and each page contains d entries, since not all the pages are fully filled, the number of loaded keys N is bounded as: $N \leq pd$.

The approximate attention scores between the query q and these N selected keys are then computed using Equation 1. The masks $s_{i,j}$ are set to 1 for the selected keys and 0 for all others.

Note that, IceCache constructs a separate DCI-tree for each attention head, allowing it to retrieve different sets of significant pages per head. This head-specific, fine-grained selection mechanism distinguishes IceCache from baselines such as Quest and ArkVale, which retrieve the same set of pages for all heads, potentially limiting their retrieval accuracy.

E. Experiment Setup

Our experimental platform comprises an Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz and an NVIDIA A100 40GB PCIe GPU. The software stack includes CUDA version 12.1, PyTorch version 2.5.1, and HuggingFace Transformers version 4.47.1. We implement IceCache on top of HuggingFace Transformers, utilizing FlashInfer for the attention kernel operation.

Algorithm 3 PAGE-SELECT

Input: Query vector $\mathbf{q}_i \in \mathbb{R}^d$, DCI-tree T, Number of critical keys k
Initialize $S_l \leftarrow \emptyset$
$S_k \leftarrow \text{QUERY}(\mathbf{q}_i, T, -1, k)$
$S_l \leftarrow \text{FIND-PAGE-INDEX}(S_k)$
Return: S_l

Algorithm 4 QUERY

Input: Query vector $\mathbf{q}_i \in \mathbb{R}^d$, DCI-tree T, Target level l, Number of critical keys k
if $l = -1$ **then**
  $l \leftarrow T$.num_level
  Set Flag $\leftarrow$ True
else
  Set Flag $\leftarrow$ False
end if
Initialize $S \leftarrow \emptyset$
Create priority queue P with capacity k
for $i = 1$ **to** l **do**
  $S' \leftarrow \emptyset$
  if $i = 1$ **then**
    $S \leftarrow \{\text{root node}\}$
  end if
  for each $s \in S$ **do**
    $S'' \leftarrow \text{Prioritized-DCI-Query}(\mathbf{q}_i, s, k)$
    $S' \leftarrow S' \cup S''$
  end for
  if Flag **or** $i = l$ **then**
    for each $s \in S'$ **do**
      Add-to-Priority-Queue(P, s)
    end for
  end if
  $S \leftarrow S'$
end for
Return: k nodes in P that have the keys with maximum inner-product with $\mathbf{q}_i$

The six tasks that we pick from LongBench are: *NarrativeQA* (answering questions based on narrative stories or scripts), *Qasper* (extracting answers from NLP research papers), *HotpotQA* (multi-hop reasoning across documents), *GovReport* (Generate long, high-quality summaries of government reports), *TriviaQA* (answering open-domain factual questions), and *Passage Retrieval* (locating relevant information from long documents). Among them, *GovReport* stands out as the most difficult task due to its requirement to generate comprehensive summaries from lengthy documents. As a long-generation task, *GovReport* poses a unique challenge in the KV-cache optimization setting because it requires sustained access to a wide range of contextual information across thousands of tokens, making eviction and recall strategies far more critical compared to short answer- or retrieval-based tasks.

We set the number of sink tokens to 32 and the number of window tokens to 32 for IceCache in all the experiments.

F. LongBench Results for LongChat-7B-v1.5

We also apply IceCache to LongChat-7B-v1.5, which is one of the most popular long-context LLMs. As shown in Table 2, IceCache consistently achieves top-tier performance on all tasks under the 256-token budget. Because PQCache does not support multi-head attention, it is not applicable. Notably, the two variants of IceCache achieve the top-2 scores across all six tasks, outperforming all baseline methods and even approaching the performance of the Full-KV oracle. In particular,

on the challenging *GovReport* task – known for its long-range generation demands – IceCache obtains a score of 29.8 and 29.8, significantly ahead of ArkVale (19.4) and RocketKV (24.6), nearly matching the Full-KV baseline (30.8). This result highlights IceCache's strength in preserving critical long-range information under tight memory budgets in the long-generation setting.

Table 2: LongBench evaluation for LongChat-7B-v1.5

Budget	Method	gov-report	narrativeqa	qasper	hotpotqa	triviaqa	pass-ret.
N/A	Full-KV	30.8	20.8	29.4	33.0	84.0	30.5
256	Exact-TopK	30.6	19.6	30.2	33.7	83.5	29.5
256	SnapKV	N/A	15.8	20.2	29.9	83.0	**28.5**
256	Quest	2.9	1.7	13.8	8.3	14.7	1.1
256	SparQ	10.4	2.5	27.3	12.9	25.4	2.5
256	RocketKV	24.6	18.8	28.6	31.0	82.5	13.5
256	PQCache	/	/	/	/	/	/
256	ArkVale	19.4	17.0	22.5	28.4	82.7	8.0
256	IceCache	29.8	**20.4**	29.5	34.6	**84.7**	**28.5**
256	IceCache (P)	**29.9**	18.9	**29.9**	33.5	84.4	27.4

G. Passkey Retrieval Accuracy

We also evaluate IceCache's effectiveness in handling long-range dependencies using the passkey retrieval task (Mohtashami & Jaggi, 2023). Following (Chen et al., 2024a), we use LongChat-7B-v1.5 as the base LLM and consider three context lengths: 10k, 20k, and 30k. For each length, 20 test cases are generated with passkeys inserted at various positions from 0% to 95% of the total context length in increments of 5%. In this section, we compare IceCache against StreamingLLM (Xiao et al., 2023), H2O (Zhang et al., 2024b), and ArkVale (Chen et al., 2024a). The results are summarized in Table 3.

Both H2O and StreamingLLM permanently evict tokens from the cache, risking the loss of passkey-relevant information. This leads to accuracies below 15%, which degrade further with increased context length or tighter cache budgets. In contrast, IceCache and ArkVale dynamically assess the importance of evicted pages and recall crucial ones on demand, consistently maintaining over 90% accuracy across all settings. Notably, IceCache's token-clustering and more precise recall mechanism allow both of its variants to achieve the same or better accuracies than ArkVale's across budget sizes and context lengths. Hence IceCache achieves a more Pareto optimal memory-accuracy trade-off.

Table 3: Accuracy of passkey retrieval tasks for LongChat-7B-v1.5

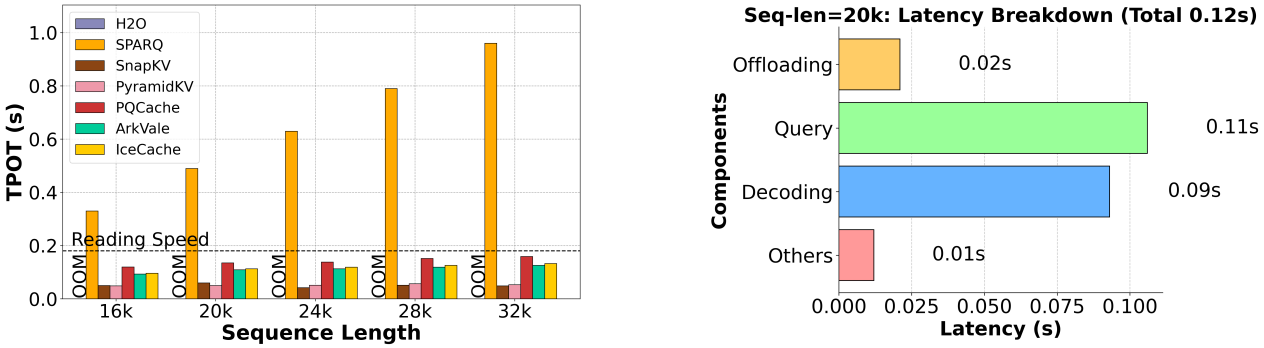
Context Length	10k				20k				30k			
Cache Budget	128	256	512	1024	128	256	512	1024	128	256	512	1024
StreamingLLM	0%	0%	0%	5%	0%	0%	0%	0%	0%	0%	0%	0%
H2O	0%	5%	5%	5%	0%	0%	0%	5%	0%	0%	5%	5%
ArkVale	100%	100%	100%	95%	100%	100%	95%	100%	90%	100%	100%	95%
IceCache	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
IceCache (P)	100%	100%	100%	100%	100%	100%	100%	100%	95%	100%	100%	100%

H. Decoding Latency Analysis

Time Per Output Token (TPOT) measures the decoding time per token during inference. As shown in Figure 6(a), we evaluate TPOT across a range of sequence lengths (16k–32k) under a unified hardware setting, with GPU caches capable of storing the keys and values for 256 tokens. Among all baselines, SPARQ exhibits the highest latency, with TPOT increasing linearly with sequence length due to its reliance on sequential query computation and communication making it unsuitable for real-time applications. H2O fails to complete the inference at all sequence lengths tested due to out-of-memory (OOM) issues. Other methods, including SnapKV, PyramidKV, PQCache, ArkVale and IceCache, show moderate latency improvements via system-level optimizations, maintaining acceptable latency even at extremely long sequence lengths. All these methods are faster than the human reading speed (indicated around 0.18s TPOT in the figure). Considering the

substantial accuracy advantage of IceCache compared to all other baselines, as previously demonstrated, we conclude that IceCache delivers superior accuracy without compromising inference speed.

Figure 6(b) presents a detailed breakdown of TPOT latency for IceCache at a sequence length of 20k, with a total latency of 0.12 s. In this figure, "Offloading", "Query", and "Decoding" represent the overhead from GPU-CPU offloading of key-value embeddings, DCI-query operations, and the overall LLM decoding process, respectively. The dominant contributor to latency is the DCI-query module (0.11s), while the original bottleneck – decoding (0.09s) – is now fully overlapped and effectively hidden behind the query process. GPU–CPU offloading and other miscellaneous operations contribute minimally, at only 0.02s and 0.01s, respectively.



(a) Time per output token.

(b) Latency breakdown.

Figure 6: The latency (TPOT) and its breakdown for IceCache (with prefetching) and the comparison methods. IceCache consistently delivers superior decoding efficiency without compromising generation quality.