RefreshKV: Updating Small KV Cache During Long-form Generation

Fangyuan Xu[1], **Tanya Goyal**[2*], **Eunsol Choi**[1*]
Department of Computer Science
[1]New York University, [2]Cornell University
{fx2145,eunsol}@nyu.edu, tanyagoyal@cornell.edu

Abstract

Generating long sequences of tokens given a long-context input is a very compute-intensive inference scenario for large language models (LLMs). One prominent inference speed-up approach is to construct a smaller key-value (KV) cache, relieving LLMs from computing attention over a long sequence of tokens. While such methods work well to generate short sequences, their performance degrades rapidly for long-form generation. Most KV compression happens once, prematurely removing tokens that can be useful later in the generation. We propose a new inference method, **RefreshKV**, that flexibly alternates between full context attention and attention over a subset of input tokens during generation. After each full attention step, we update the smaller KV cache based on the attention pattern over the entire input. Applying our method to off-the-shelf LLMs achieves comparable speedup to eviction-based methods while improving performance for various long-form generation tasks. Lastly, we show that continued pretraining with our inference setting brings further gains in performance.

1 Introduction

Large language models (LLMs) are capable of ingesting extremely long inputs and generating long outputs (Meta, 2024; Gemini, 2024). Yet, deploying such long-context LLMs is very costly. As the context length increases, memory usage for storing the key-value (KV) cache increases linearly, while attention computation scales quadratically. These two factors lead to high latency during inference; Adnan et al. (2024) reports 50x latency increase as context length increased 16x for the MPT-7B model (MosaicML, 2023).

Prior works (Beltagy et al., 2020; Child et al., 2019; Xiao et al., 2023; Zhang et al., 2024b; Li et al., 2024; Adnan et al., 2024) propose to maintain a smaller KV cache by evicting a subset of past tokens. These approaches improve both the memory and computation efficiency, as the KV cache of only a subset of tokens will be kept and attention computation is reduced. However, once an input token is eliminated from the KV cache (either based on locality assumption (Xiao et al., 2023) or by eviction during the generation process (Zhang et al., 2024b)), one cannot recover eliminated tokens. We find that while such methods show minor degradation compared to full KV cache in short-form generation tasks, their performance degrades rapidly for long-form generation tasks.

Having observed the limitations of existing approaches, we propose a novel approach, **RefreshKV**, which periodically refreshes the smaller KV cache during the generation process. Our method keeps the full KV cache throughout inference (thus no gain in memory footprint), but perform attention over a dynamically constructed small KV cache to achieve inference speedups. Our method alternates between two modes of generation: generation that attends over the full KV cache and generation that attends over a smaller KV cache with subset of tokens (see Figure 1). To construct the smaller KV, we identify the topK attended tokens from the most recent step that attends over the full KV cache, observing that consecutive tokens have similar attention pattern (Li et al., 2024).

A key component of RefreshKV is deciding when to perform the computationally expensive full attention steps and refresh the small KV cache. Instead of mandating a fixed (and potentially suboptimal) schedule, RefreshKV compares the query embedding similarity of the current and previous full attention step, and dynamically triggers full attention step when the similarity is low. Our approach (no KV eviction, dynamically constructed smaller KV, low latency) establishes a middle ground between full attention (no KV eviction, high latency,

*Equal advising.

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 24878–24893
July 27 - August 1, 2025 ©2025 Association for Computational Linguistics

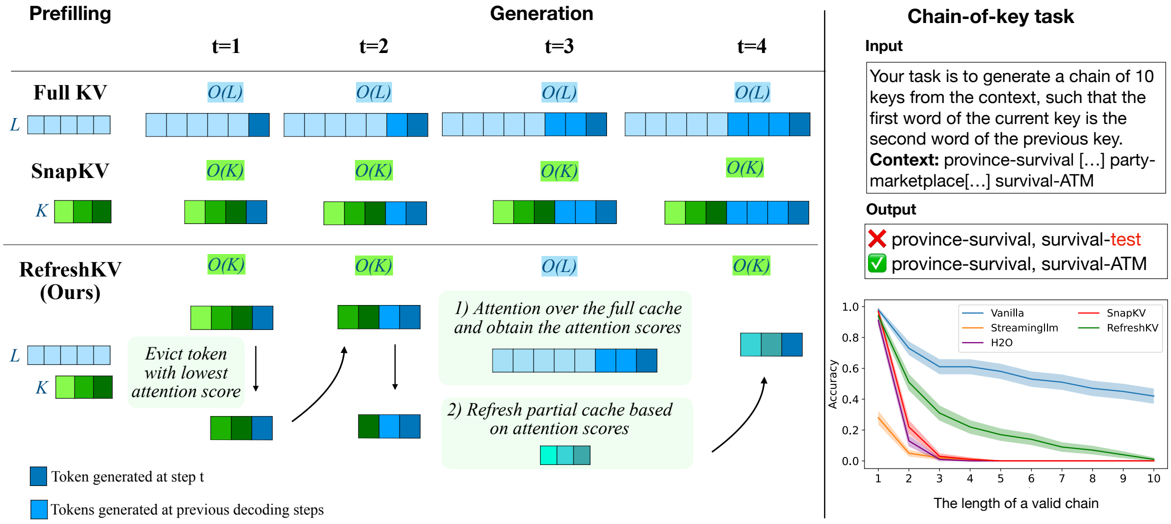

Figure 1: Left: Illustration of **RefreshKV** (with $L = 5$, $K = 3$ and a stride $S = 3$) compared to baseline (SnapKV and Full KV) when generating four tokens. The figure shows the computation complexity of attention operation, and the size of the KV cache used at each decoding step for each method. Our approach alternates between inferencing with the partial cache(t=1,2,4) and the full cache(t=3). Compared to eviction-based method (e.g. SnapKV) which completely discard the evicted tokens, **RefreshKV** updates the partial cache based on attention scores over the entire context during the full attention steps. Right: An example of the chain-of-key task and performance of **RefreshKV** and the baselines. RefreshKV maintains performances across different length while eviction-based baselines' performance degrades when generating a chain with more than one key.

high performance) and sparse attention (KV eviction, reduced latency, low performance), particularly useful for long-form generation.

Our method can be applied to any off-the-shelf LLM. We experiment with two long-context LLMs, Llama-3.1-8B (Meta, 2024) and Qwen2-7B (Yang et al., 2024a). We compare against KV eviction baselines StreamingLLM (Xiao et al., 2023), H_2O (Zhang et al., 2024b) and SnapKV (Li et al., 2024) on the long-range language modeling task and a suite of downstream long-context tasks (Bai et al., 2023; Zhang et al., 2024a; Ye et al., 2025) that require long outputs given long inputs.

Our experiments show that **RefreshKV** outperforms eviction-based methods in both these settings, with similar level of speed-up. In particular, we examine two long-form generation tasks that are not evaluated by previously proposed eviction-based methods: (1) when majority of tokens are required to generate the output (e.g. converting information in an HTML page to a TSV file) and (2) when the important tokens required at the current generation step is dependent on the previously generated tokens (a new task, **Chain-of-key**, as depicted in Figure 1). While eviction-based methods such as H_2O and SnapKV fail completely in HTML to TSV task (Ye et al., 2025), achieving 0 F1 score, RefreshKV recovers 52% of the performance. Our analysis shows that the performance gains are attributed to updating the partial cache rather than occasionally attending to the entire output. Lastly, we explore continued pretraining Llama-3.1-8B with RefreshKV, which leads to further improvements. Our contributions are as follows:

- We identify the failures of existing KV eviction methods when LLMs are tasked with challenging long-form generation tasks. To reveal this weakness, we propose a new task (**Chain-of-key**) which requires remembering the input context more comprehensively.
- Motivated by the failures of KV cache eviction methods, we introduce a new inference method, **RefreshKV**, that rebuilds a smaller KV cache periodically during long-form generation.
- We evaluate our method comprehensively on various benchmarks and two LLMs, and conduct ablation studies on our design choices (e.g., dynamic stride vs. fixed stride cache updates).

We release our code and the chain-of-key dataset at `https://github.com/carriex/refreshkv`.

2 RefreshKV for Long-Form Generation with Long-Context LLMs

2.1 Background and Setting

Let M be a language model and $\mathtt{x}$ be an input sequence of tokens, $x = x_1, \cdots x_L$. At inference time, M generates an output token sequence $\hat{y} = y_1, \cdots y_N$ in two stages: (1) **Pre-filling stage** where M ingests the input and constructs the KV cache for all L tokens, and (2) **Generation stage** where it samples one token y_i at a time from the conditional distribution $P_M(y_i|x, y_1 \cdots y_{i-1})$. At each step, the model attends to tokens in the KV cache, and updates the cache to include the current token's key-value pairs.

Our goal is to reduce the inference latency during the generation stage without severe degradation of model performance. There are two main reasons for latency increase; first, the attention computation increases quadratically with input length L. Second, a large L necessitates maintaining a large KV cache of the past tokens, incurring latency due to the full KV cache movement from the GPU HBM.[1]

Prior approaches, like H_2O (Zhang et al., 2024b) and SnapKV (Li et al., 2024), address this by permanently evicting "unimportant" tokens during the decoding process to maintain a small KV cache. While such methods have shown to be effective for short-form generation task such as "Needle-in-a-Haystack"(NIAH) (Kamradt, 2023), it has the potential downside of prematurely removing tokens useful for subsequent generation steps. Instead of this strict strategy, we propose to periodically *update* the small KV cache by performing full attention over all the tokens in the context and constructing the small cache based on the attention pattern. As the cache is only occasionally updated, our method reduces both attention computation and data movement by attending to the small cache.

2.2 Methodology and Implementation

We present the pseudocode for generating output tokens using **RefreshKV** in Figure 2. The algorithm takes as input a language model M and a sequence of input tokens $x_1, ..., x_L$. As a first step, we prefill M with the input sequence. Then, we alternate between full and partial attention. Our approach maintains two separate KV caches C_f and C_p, corresponding to KV cache used in the full and partial attention steps respectively. These three components of the algorithm are described below:

Prefilling stage (lines 1-2): Given input $x_1, ..., x_L$, we prefill with full attention M and initialize full KV cache C_f with L tokens. We also obtain the attention scores $\mathbf{a_L}$ for the last token x_L. To determine the top K tokens to keep, we employ max pooling over attention scores of surrounding tokens, instead of the raw attention scores to preserve information completeness following prior work (Li et al., 2024).[2]

Deciding when to decode with full cache (line 4): We need to decide when to alternate between performing attention over all tokens and performing attention over the smaller cache. One straightforward way is to use a fixed schedule, i.e. performing full attention every S steps. However, this enforces the same schedule for *all* the layers and input text. Instead, we propose an adaptive schedule based on the similarity between query vector of the current step and the query vector of the most recent full attention step. Intuitively, if the query vector of a particular layer and head for the current step is similar to the query vector of the most recent full attention step, the attention pattern should be similar. Thus, we only perform the full attention step when this similarity is lower than a threshold.

Concretely, at every S^{th} decode step, for each layer l, we first determine whether we *need* to perform full attention. We calculate the cosine similarity between the query vectors of the input token t averaged across all query heads in layer l, with the averaged query vector of the most recent full attention step for that layer. If the similarity is higher than a threshold s, we decode with the partial cache C_p, and otherwise decode with C_f for layer l. We describe details for each scenario below. To minimize the computational overhead of the similarity check, we perform this only every S steps; we call this query comparison (QC) stride.

Decoding with partial cache (lines 5-7): At each partial attention step, we generate the next token $y_t \sim M(C_p)$ using C_p to compute attention and store the KV cache of the input token. This leads to a reduction in both the attention computation FLOPs and the latency due to KV cache movement (we only need to move the smaller KV cache C_p instead of the larger full KV cache C_f, where $|C_p| << |C_f|$). To maintain the size of C_p as we decode each additional token and update the KV cache with this newly generated token, we remove the KV corresponding to the token with

[1] Adnan et al. (2024) reports up to 40% of the inference latency can be attributed to data movement.

[2] For models with Grouped Query Attention (Ainslie et al., 2023), we aggregate attention scores for all query heads in the same group by taking the max to identify the top K tokens. Our ablations (reported in Table 8 in the Appendix) show that taking the max outperforms other aggregation method such as mean, or relying solely on one of the query head in the group.

Algorithm 1: Generation using RefreshKV

Input: Language model M, input $x_1, \ldots, x_L$
Hyperparameters: partial cache size K, a scheduling strategy S
Output: Sequence of generated tokens O

Prefilling

1. Prefill M with $x_1, \ldots, x_L$
    1. Initialize full KV cache C_f
    2. $a_L \leftarrow$ attention scores for $x_L$over past tokens for all layers.
    3. Initialize partial KV cache $C_p \leftarrow$ reverse(arg top-k$_{x \in C_f}$(max_pool($\mathbf{a}_L$, kernel_size)))
    4. $pos_{full} \leftarrow 1$ to keep track of last token decoded with C_f
2. $O \leftarrow []$ // Initialize empty output sequence

Decoding

3. **for** $i \in 1,2,\cdots,T$ **do**
4. **if scheduler-S**(i) == "partial-attention-step" **then**
5. $o \leftarrow M(C_p)$, O.append(o) // Generate using partial cache
6. $C_p \leftarrow [C_p; \mathrm{KV}(x_{L+i})]$ // Update partial cache C_p
7. $C_p \leftarrow C_p[1:]$ // Evict token with lowest **a** from C_p
8. **else-if scheduler-S**(i) == "full-attention-step" **then**
9. $C_f \leftarrow [C_f; C_p[-(i - pos_{full}):]]$ // Update C_f with *new* tokens in C_p
10. $o, \mathbf{a} \leftarrow M(C_f)$, O.append(o) // Generate using full attention
11. $C_f \leftarrow [C_f; \mathrm{KV}(x_{L+i})]$ // Update full cache C_f
12. $C_p \leftarrow$ reverse(arg top-k$_{x \in C_f}$(max_pool($\mathbf{a}_L$, kernel_size))) // Re-initialize C_p with topK
13. $pos_{full} \leftarrow i + 1$ // Set pos_{full}
14. **return** O

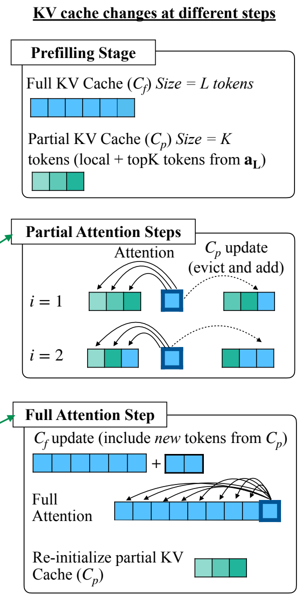

Figure 2: Pseudocode for **RefreshKV**. The model prefills the prompt with full attention and initialize the partial cache C_p cache with attention scores of the last token. For each partial attention step, we decode with the partial cache and append the KV pairs of the input token to the partial cache. We evict the token with the lowest attention score to maintain a fixed-sized partial cache. For the full attention step, we first update the full KV cache with the new tokens decoded with the partial cache, then decode with the full cache and refresh the partial cache.

the lowest attention score in the full attention step from C_p (line 7). We note that decoding with C_p is equivalent to SnapKV (Li et al., 2024) if the partial cache is never refreshed after prefilling.

Decoding with full cache (lines 9-13): At each full attention step, we first update the full KV cache C_f with the key-value pairs of the tokens decoded with C_p. Next, we generate the next token $y_t \sim M(C_f)$ using the full KV cache C_f and obtain the attention scores $\mathbf{a_L}$. Finally, we refresh the partial cache C_p with the topK tokens based on $\mathbf{a_L}$.

Memory and Time requirements Our method has memory requirement similar to that of vanilla attention, as we are not permanently evicting any tokens from the KV cache. However, our decoding latency is on par with other KV cache eviction methods, as later shown in our experiments in Section 3. We discuss memory and speed considerations in detail in Appendix A.2.

3 Experiment Setup

Models and Evaluation tasks We evaluate our method with two long-context language models Llama-3.1-8B (Meta, 2024) and Qwen2-7B (Yang et al., 2024a). Both models can process inputs of up to 128K tokens. We conduct experiments on language modeling and downstream tasks:

- **Language modeling** We measure perplexity of the Arxiv and Book split of RedPajama (Together, 2023) with context size of 16K. We report results on 100 sequences for each domain. To simulate long-form generation, we report the perplexity of the last 256 tokens.

- **Long-input, short output tasks:** We report the performance of RULER (Hsieh et al., 2024), which consists of a set of 13 tasks with context size of 32K that require short output.

- **Long-input, long output tasks** We evaluate our methods on three sets of downstream tasks which require the model to generate long-form outputs (more than 100 tokens) given long-form inputs (more than 10k tokens).[3] (1) **long-context summarization tasks**: QMSum (Zhong et al., 2021), GovReport (Huang et al., 2021) and Novel Summarization (Zhang et al., 2024a) and (2) **HTML to TSV** task from LongProc (Ye et al., 2025) benchmark.[4] We report results aggregated across

[3] For each dataset, we filter examples with input length <10K tokens. We report dataset statistics for each dataset in Table 6 in the Appendix.

[4] We exclude the other tasks from LongProc as they primarily involve short inputs, resulting in minimal speedup in our

three output lengths (0.5K, 2K, and 8K). We report ROUGE-L for summarization tasks and row-level F-1 score for the **HTML to TSV** task.

- **New task: Chain-of-key generation** We propose a synthetic task where model's previous generation steps, together with its long context input, guides future generation steps. Given a context which consists of a list of two-word keys, the model is tasked with generating a sequence of T keys, such that the first word of the next key is the last word of the current key. This task requires models to look up information in the context based on what has been previously generated, resembling multi-hop retrieval. An example of the task is illustrated in Figure 1. We report accuracy of the output by the relative length of a valid chain (i.e. the length of the valid sub-chain divided by T). More details and examples are in Section A.6 in the Appendix.

Comparison systems We implement the following baselines: (1) ***Vanilla*** attention that maintains and performs attention over the full KV cache (2) ***StreamingLLM*** (Xiao et al., 2023) which consists of "sink tokens" and recent tokens. (3) ***H*$_2$*O*** (Zhang et al., 2024b) which consists of recent tokens and dynamically updated "heavy hitters", defined by high cumulative attention scores. (4) ***SnapKV*** (Li et al., 2024) which consists of tokens with high attention scores from the last few tokens in the prompt. We describe the setting for each baseline in Section A.1 in the Appendix.

Inference settings We prefill the model with the input and report wall clock times for the decoding phase. Our experiments are run on a single A100 80GB GPU using Flash Attention (Dao, 2024).[5] We set K to be 1/8 of the input length. NovelSumm contains the longest input length (100K tokens) and we set K to be 4096, corresponds to $1/25L$. We report results with greedy decoding. For RefreshKV, we report results for two different query comparison strides $\{5, 10\}$ with a similarity threshold s of 0.85 for Llama-3.1-8B and 0.95 for Qwen2-7B. We determine the value of s by experimenting with a range of values on a held-out set of the Book dataset (reported in Section A.4 in the Appendix) and apply the same threshold for all the tasks.

setting. For completeness, we report the performance of these tasks in Section A.7 in the Appendix, observing a similar trend as the HTML to TSV task in terms of end-task performance.

[5]We describe implementation details in Section A.1.

Method	Arxiv/Book PPL ↓	Time ↓
	Llama-3.1-8B	
Vanilla	2.22/7.07	7.50
Streaming	2.62/7.94	6.61
H_2O	2.48/7.60	10.77
SnapKV	2.54/7.78	6.77
Refresh (QC=5)	**2.27/7.31**	6.67
Refresh (QC=10)	2.32/7.41	6.33
	QWEN-2-7B	
Vanilla	2.33/8.26	9.07
Streaming	2.75/9.10	6.27
H_2O	2.68/9.02	11.57
SnapKV	2.80/9.18	6.09
Refresh (QC=5)	**2.39/8.55**	6.71
Refresh (QC=10)	2.49/8.72	6.33

Table 1: Perplexity results and latency on language modeling task for LLama-3.1-8B and QWEN-2-7B. We report results on Arxiv and Book corpora with input context length of $16K$ tokens. We set $K = 2048$.

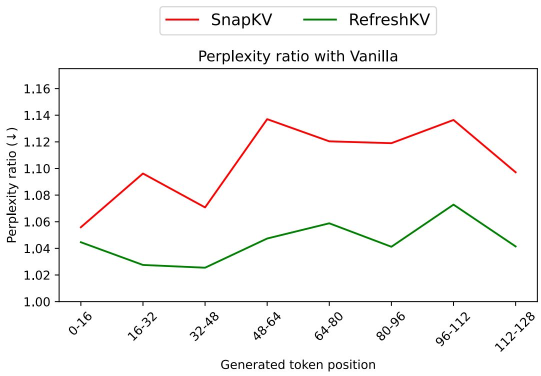

Figure 3: We plot the perplexity ratio against the vanilla baseline for RefreshKV (with stride of 10) and SnapKV based on the tokens generated (x axis). While the ratio is similar at the beginning of the sequence, as the generation goes SnapKV's perplexity diverges from vanilla approach while that of RefreshKV is relatively stable.

4 Results

4.1 Language Modeling

Table 1 outlines the performance of the baselines and RefreshKV for perplexity. For both models, RefreshKV achieves better perplexity and comparable inference speeds compared to StreamingLLM and SnapKV for $QC = 10$. Our method also achieves better performance than the best baseline, H_2O, with a much shorter inference time per example, as we do not require accessing attention score at each decoding step. Setting $QC = 5$ increases inference time but also brings performance gain compared to $QC = 10$, allowing a performance-efficiency trade-off.

The key distinction between RefreshKV and SnapKV is that our method *refreshes* the partial

Input/Output length	32K/<30	10K/ 0.1K	10K/0.7K	128K/1K	30K/2.2K	22K/50
Dataset	**RULER**	**QMSum**	**GovReport**	**NovelSumm**	**HTML to TSV**	**Chain-of-key***
Method	**Acc↑**	**R-L↑**	**R-L↑**	**R-L↑**	**F-1↑**	**Acc↑**
Vanilla	90 / 79	25.63 / 24.98	34.11 / 33.38	31.29 / 19.91	33 / 24	56 / 83
Streaming	22 / 21	22.27 / 20.30	16.30 / 23.84	24.66 / **22.11**	2 / 5	2 / 2
H_2O	21 / 21	22.12 / 20.83	27.41 / 26.91	19.31 / 18.51	0 / 0	10 / 11
SnapKV	79 / 58	24.33 / 22.93	28.06 / 28.80	29.23 / 19.09	0 / 0	12 / 13
RefreshKV (QC=5)	**86 / 75**	**24.92 / 24.34**	**32.56 / 31.40**	**29.98** / 19.70	**17 / 10**	**25 / 24**
RefreshKV (QC=10)	80 / 67	24.73 / 23.98	31.47 / 31.36	29.37 / 18.94	8 / 6	15 / 15

Table 2: Downstream task performance. In each cell, the first number represents the performance of Llama-3.1 model and the second number for QWEN-2 model. *We report performance of Llama-3.1-70B and Qwen-2-72B for the chain-of-key task, as the smaller variants cannot perform the task even in vanilla setting.

cache as generation progresses. We compare the perplexity degradation ratio of both methods relative to vanilla attention over different generation timestamps in Figure 3 with Llama-3.1-8B on the book dataset. While both methods begin with a similar perplexity ratio compared to vanilla (step 0-16), SnapKV's performance degrades as generation proceeds, whereas RefreshKV maintains a stable ratio, highlighting the benefit of refreshing the small KV cache during generation.

4.2 Downstream Tasks

Results for downstream tasks are reported in Table 2. We also report the average input and output length for each dataset. For RULER, we report results aggregated over 13 tasks here and report the per-task performance in Table 14 in the Appendix.

Eviction-based methods fail for long-form generation tasks. Baseline methods that evict tokens from the KV cache permanently (StreamingLLM, H_2O and SnapKV) show degradation for tasks that require long-form outputs. While SnapKV performs better than the other two baselines on RULER, it shows severe performance degradation on the HTML to TSV task, achieving 0 F-1 scores for the former. For the Chain of key task, eviction-based methods are unable to generate a chain with more than two keys, achieving accuracy < 20.

RefreshKV closes the gap between vanilla and eviction-based approach. On HTML to TSV task, RefreshKV with $QC = 5$ recovers 52% and 42% of performance for Llama-3.1-8B and Qwen2-7B respectively. On the **Chain-of-key** task, RefreshKV is the only method that is able to generate a valid key with length longer than two keys, as shown in Figure 1. For the long-form summarization tasks, RefreshKV outperforms baselines in all three datasets, except for NovelSumm with Qwen2-7B, where StreamingLLM outperforms the vanilla full attention.[6] We also observe gains for RULER tasks, particularly the subtasks that require generating longer output (e.g. generating multiple keys), which we discuss in Section A.8 in the Appendix.

5 Ablation Studies

5.1 Adaptive stride vs. Fixed stride

We trigger full attention step when the query vector of the input token is substantially different from the query vector of the most recent full attention step. Can we use a simpler strategy to decide when to perform full attention? In this section, we explore refreshing at a fixed stride, performing full attention every N-th step across all the layers.

Setting We compare the results of (1) employing a dynamic stride with the set-up in Section 3, i.e. QC stride of {5, 10} and similarity threshold $s = 0.85$ (Llama-3.1-8B) and $s = 0.95$ (Qwen2-7B) and (2) employing a fixed stride S of {10, 15} for comparable decoding time. We report results on the language modeling task on the Book dataset and two downstream tasks. We report the decoding time measured on one A100 machine. For the language modeling task, we report the time for generating 256 tokens. For the downstream tasks, we measure the time of generating the first 50 tokens. We also report the *effective* stride averaged across all the layers, i.e. how often is full attention performed when employing dynamic strides.

Results Table 3 presents the results. For Llama-3.1-8B, comparing $QC = 5$ and $S = 10$, employing dynamic stride consistently achieves better

[6]We hypothesize that this might be due to the fact that Qwen is pre-trained on 32K context and adopts YARN and Dual Chunk Attention to enable processing of up to 128K tokens. As NovelSumm contains > 100K tokens, it exceeds the pre-training context window, which might cause the performance difference between StreamingLLM (which uses local window) and other methods that leverage attention scores over full context (including vanilla).

Schedule	Book			HTML (0.5K)			GovReport		
	Time ↓	Stride	PPL ↓	Time ↓	Stride	Acc ↑	Time ↓	Stride	R-L ↑
Llama-3.1-8B									
Vanilla	7.50	-	7.07	1.52	-	43	1.43	-	34.11
Fixed	7.20	10	7.40	1.40	10	17	1.37	10	32.30
Dynamic (QC=5, s=0.85)	7.17	12	**7.31**	1.40	14	**30**	1.38	14	**32.56**
Fixed	6.99	15	7.45	1.37	15	8	**1.34**	15	30.67
Dynamic (QC=10, s=0.85)	**6.89**	17	7.41	**1.33**	19	16	**1.34**	19	31.47
Qwen-2-7B									
Vanilla	9.07	-	8.26	1.96	-	35	1.73	-	33.38
Fixed	6.59	10	8.74	1.38	10	8	1.29	10	31.18
Dynamic (QC=5, s=0.95)	6.71	7	**8.55**	1.29	7	**20**	1.34	7	**31.40**
Fixed	6.43	15	8.81	1.31	15	9	**1.27**	15	30.73
Dynamic (QC=10, s=0.95)	**6.33**	11	8.72	**1.23**	12	14	1.28	12	31.36

Table 3: Results comparing fixed stride and dynamic stride based on query similarity. In all tasks, dynamic stride shows better task performance while performing full attention step fewer times.

Method	Stride	Arxiv	HTML (0.5K)
Vanilla	-	2.22	43
SnapKV	-	2.54	0
RefreshKV	10	2.32	16
- w/o refresh	10	2.50	0
- w/o full attention	10	2.32	16

Table 4: Ablation study on LLama-3.1-8B. We report perplexity for Arxiv and F-1 score for the HTML to TSV task.

performance with similar or less decoding time for all three tasks. We see a similar trend comparing $QC = 10$ and $S = 15$. For Qwen2-7B, dynamic stride achieves better performance across all three tasks, with slightly more decoding time on Govreport. We also observe slightly different effective stride for different tasks when employing the same QC and s, showing that dynamic stride enable flexible scheduling based on the context. We report per-layer stride in Section A.5 in the Appendix.

5.2 Impact of full attention steps

Compared to other baseline methods which never perform full attention during the generation, RefreshKV involves extra attention calculation (i.e. attending over the entire output). To tease apart the performance gains from occasional full attention step and updating the small KV, we present two ablation setting for RefreshKV: (1) *w/o refresh* which performs attention over the full KV cache at the fixed stride of S but without refreshing the partial cache. This is equivalently using the partial cache obtained with SnapKV and occasionally performing full attention. (2) *w/o full attention* which calculates the attention scores over the entire KV cache and updates the partial cache, then attends to the updated partial cache, instead of attending to the full KV cache, at stride S.

Results are in Table 4. While performing occasional full attention (**w/o refresh**) improve perplexity slightly compared to SnapKV, the performance lags behind RefreshKV. In contrast, the ablation setting where partial cache is refreshed (**w/o full attention**) achieves the same performance of RefreshKV for both tasks. This shows that the gain of RefreshKV mostly comes from refreshing the partial KV cache, instead of performing occasion full attention over the entire cache.

6 Continued Pre-training with RefreshKV

We have demonstrated **RefreshKV** can be used as an inference-time method. However, since the LLMs we study are trained with full attention, applying RefreshKV during inference introduces a discrepancy between training and inference. Specifically, it involves attending to a non-contiguous sequence of tokens in the partial cache. Here, we explore continued pretraining with RefreshKV to adapt models to this new attention pattern.

To make training setting simpler, we do not fully implement RefreshKV during training. We use a fixed stride of 50 and never refresh the partial cache. We assume a length $L+S$ for all sequences, where L is the pre-fill length. We perform standard attention over all past tokens for the first L tokens. We emulate the partial attention pattern for the last S tokens in the sequence during training. For the next S tokens, we perform attention over the top K tokens identified as well as local tokens (i.e. tokens $L+1$ onwards). We train the model with next token

Method	Stride	Test PPL (8K)	Test PPL (16K)
Vanilla	-	2.70 → 2.70	2.50 → 2.50
Streaming	-	3.40 → 3.38	3.50 → 3.49
H_2O	-	3.95 → 3.90	3.52 → 3.49
SnapKV	-	3.21 → 3.15	2.98 → 2.92
RefreshKV	10	2.83 → 2.79	2.57 → 2.56
RefreshKV	25	2.97 → 2.93	2.67 → 2.63
RefreshKV	50	3.13 → 3.05	2.79 → 2.72

Table 5: Results on continued pre-training with RefreshKV for LLaMA-3.1. The context size is 8k and we report perplexity on the last 50 tokens. We report the performance for each setting before (the number on the left) and after (the number on the right) CPT.

prediction loss for all the tokens in the sequence.

Setup We set $L = 8092$, $S = 50$ and $K = 2048$ for this experiment. We randomly sample a subset of 200k sequences from the Arxiv split of RedPajama dataset. We split the data into 80%, 10% and 10% train/dev/test splits, resulting in 120k training data samples. We perform continued pre-training on Llama-3.1-8B and describe implementation details in Section A.1 in the Appendix.

Evaluation As our continued pretraining is relatively small scale on the base model, we focus on evaluating on the language modeling task for two settings: (1) input size $L =$8K consistent with the training set-up and (2) $L =$16K. We set $K = 1/8L$ For each method, we report the performance from the pre-trained checkpoint and the performance after continued pre-training.

Results We report the results in Table 5, each row represents a different inference strategy on the same model. Despite the mismatch in how partial KV was constructed, continued pre-training benefits other methods (Streaming, H_2O) slightly. We see larger gain for RefreshKV from continued pre-training across all settings. Our training assumes a fixed stride of 50, but we see performance gain for different strides ($S = 10, 25$). Training on shorter context (8K) also translates to gains when inferencing on longer context (16K), showing promise for improving the performance of RefreshKV with continued pre-training.

7 Related Work

Efficient inference methods Various techniques have been proposed to enhance inference efficiency, which are orthogonal to and can be combined with our approach. FlashAttention (Dao, 2024) achieves significant gain in inference speed by optimizing attention computations on GPUs. A line of work (Xiao et al., 2022; Liu et al., 2024b; Hooper et al., 2024) proposes to quantize KV caches to reduce both memory and computation cost; while recent work (Liu et al., 2024a; Wan et al., 2025; Wang et al., 2025) proposes to merge KV cache of similar tokens.

KV cache eviction Recent work extensively studies KV cache eviction strategies, such as keeping only "sink" and recent tokens in the KV cache (Xiao et al., 2023); or tokens with high accumulative attention scores (Zhang et al., 2024b). A line of work propose query-aware eviction strategies, using the attention scores of the last few tokens in the prompt to select tokens to keep (Li et al., 2024; Chen et al., 2024). Other works design eviction strategies based on attention patterns of different heads (Ge et al., 2024; Xiao et al., 2024b) or different layers (Cai et al., 2024; Yang et al., 2024b; Wan et al., 2025). We show that such eviction-based methods can fail on long-form generation tasks and propose to refresh the small KV cache during generation.

Sparse attention Our method achieves efficiency by performing sparse attention. Earlier work (Zaheer et al., 2020; Beltagy et al., 2020) investigates training LLMs with a fixed sparse attention pattern (such as a sliding window) to reduce computational complexity. Training-free methods such as Unlimiformer (Bertsch et al., 2023) and InfLLM (Xiao et al., 2024a) performs attentions on subset of tokens which received the highest attention scores, with the goal of extending the context window of a given language model. In contrast, we leverage previous tokens' attention scores to select tokens to attend to for long-context models, which can already handle sequences with up to 128k tokens. MInference (Jiang et al., 2024) identify head-specific patterns to perform sparse attention, focusing on accelerating the prefilling stage. Similar to ours, SparQ (Ribar et al., 2024) and Quest (Tang et al., 2024) achieves decoding time speed-up by attending to subset of tokens. Instead of leveraging the attention patterns of previous tokens, these methods build specialized kernel to approximate attention and identify critical tokens.

8 Conclusion

We propose **RefreshKV**, an inference-time method which accelerates long-form generation for long-context input by decoding from a small, dynamic KV cache that is updated based on attention patterns of neighboring tokens. Compared to previous work which permanently evict tokens from the context, **RefreshKV**maintains the full KV cache and alternates between inferencing over the full and small KV cache. We apply our method to two off-the-shelf long-context models and show that our method reduces inference wall-clock time while better preserving performance compared to eviction-based methods on long-form generation tasks. Finally, we show that continued pre-training the model with **RefreshKV** can further improve the performance-efficiency trade-off.

Limitations

Proposed method While we focus on accelerating inference speed, our method does not reduce memory requirement for using long-context LLMs, which can be a bottleneck for certain use cases. Our objective is to accelerate decoding for long-context models. While our method outperforms eviction-based approaches, it still involves a trade-off between performance and efficiency. In this study, we employ query similarity based dynamic scheduling to decide when to perform full attention and refresh the small KV cache. Future work can explore other strategy, such as more exhaustively tuning the similarity threshold, or setting a different threshold per layer.

Experimental settings We have conducted experiment with two open-sourced long-context models and two evaluation tasks setting. We did not test out more language models and other long-context benchmarks (An et al., 2023; Karpinska et al., 2024) given our limited compute resources. For the same reason, our experiment on continued pre-training is relatively small scale on a limited domain. We have demonstrated the effectiveness of refreshing a small KV cache constructed with attention scores and use the same size across different layers. Future work can extend our method to refresh smaller cache constructed with different strategy, e.g. layer-specific strategies (Yang et al., 2024b; Cai et al., 2024). Finally, our method is not limited to the language domain. Future work can explore applying RefreshKV to other modalities, for example, vision transformers.

Acknowledgments

We thank Xi Ye, Wenting Zhao and the UT NLP group for helpful feedback. We also thank reviewers for helpful feedback for an earlier manuscript. The work is partially supported by a gift from Apple and NSF grant RI-2312948. This work was done in part while the first and last author was visiting the Simons Institute for the Theory of Computing.

References

Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. *Proceedings of Machine Learning and Systems*, 6:114–127.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebr'on, and Sumit K. Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *ArXiv*, abs/2305.13245.

Chenxin An, Shansan Gong, Ming Zhong, Xingjian Zhao, Mukai Li, Jun Zhang, Lingpeng Kong, and Xipeng Qiu. 2023. L-eval: Instituting standardized evaluation for long context language models. *Preprint*, arXiv:2307.11088.

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. 2023. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv:2004.05150*.

Amanda Bertsch, Uri Alon, Graham Neubig, and Matthew Gormley. 2023. Unlimiformer: Long-range transformers with unlimited length input. In *Advances in Neural Information Processing Systems*, volume 36, pages 35522–35543. Curran Associates, Inc.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, et al. 2024. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *CoRR*.

Yilong Chen, Guoxia Wang, Junyuan Shang, Shiyao Cui, Zhenyu Zhang, Tingwen Liu, Shuohuan Wang, Yu Sun, Dianhai Yu, and Hua Wu. 2024. NACL: A general and effective KV cache eviction framework for LLM at inference time. In *Proceedings* of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7913–7926, Bangkok, Thailand. Association for Computational Linguistics.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *ArXiv*, abs/1904.10509.

Tri Dao. 2024. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*.

Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 2021. 8-bit optimizers via block-wise quantization. *CoRR*, abs/2110.02861.

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2024. Model tells you what to discard: Adaptive KV cache compression for LLMs. In *The Twelfth International Conference on Learning Representations*.

Gemini. 2024. Google. gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*.

Luyang Huang, Shuyang Cao, Nikolaus Parulian, Heng Ji, and Lu Wang. 2021. Efficient attentions for long document summarization. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1419–1436, Online. Association for Computational Linguistics.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *arXiv preprint arXiv:2407.02490*.

Gregory Kamradt. 2023. Needle in a haystack - pressure testing llms, commercially usable llms.

Marzena Karpinska, Katherine Thai, Kyle Lo, Tanya Goyal, and Mohit Iyyer. 2024. One thousand and one pairs: A "novel" challenge for long-context language models. *Preprint*, arXiv:2406.16264.

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. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Gholamreza Haffari, and Bohan Zhuang. 2024a. Minicache: KV cache compression in depth dimension for large language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024b. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *ArXiv*, abs/2402.02750.

Meta. 2024. The llama 3 herd of models. *ArXiv*, abs/2407.21783.

NLP Team MosaicML. 2023. Introducing mpt-7b: A new standard for open-source, commercially usable llms.

Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. 2024. SparQ attention: Bandwidth-efficient LLM inference. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 42558–42583. PMLR.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 2024. Quest: Query-aware sparsity for efficient long-context llm inference. *Preprint*, arXiv:2406.10774.

Together. 2023. Redpajama: an open dataset for training large language models.

Zhongwei Wan, Xinjian Wu, Yu Zhang, Yi Xin, Chaofan Tao, Zhihong Zhu, Xin Wang, Siqi Luo, Jing Xiong, Longyue Wang, and Mi Zhang. 2025. D_{2}O: Dynamic discriminative operations for efficient long-context inference of large language models. In *The Thirteenth International Conference on Learning Representations*.

Zheng Wang, Boxiao Jin, Yuming Chang, Zhongzhi Yu, and Minjia Zhang. 2025. Model tells you where to merge: Adaptive KV cache merging for LLMs on long-context tasks.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, Song Han, and Maosong Sun. 2024a. Infllm: Unveiling the intrinsic capacity of llms for understanding extremely long sequences with training-free memory. *arXiv*.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Julien Demouth, and Song Han. 2022. Smoothquant: Accurate and efficient post-training quantization for large language models. *ArXiv*, abs/2211.10438.

Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. 2024b. Duoattention: Efficient long-context llm inference with retrieval and streaming heads. *arXiv*.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *ArXiv*, abs/2309.17453.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Ke-Yang Chen, Kexin Yang, Mei Li, Min Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yunyang Wan, Yunfei Chu, Zeyu Cui, Zhenru Zhang, and Zhi-Wei Fan. 2024a. Qwen2 technical report. *ArXiv*, abs/2407.10671.

Dongjie Yang, Xiaodong Han, Yan Gao, Yao Hu, Shilin Zhang, and Hai Zhao. 2024b. PyramidInfer: Pyramid KV cache compression for high-throughput LLM inference. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 3258–3270, Bangkok, Thailand and virtual meeting. Association for Computational Linguistics.

Xi Ye, Fangcong Yin, Yinghui He, Joie Zhang, Yen Howard, Tianyu Gao, Greg Durrett, and Danqi Chen. 2025. Longproc: Benchmarking long-context language models on long procedural generation. *arXiv preprint*.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big bird: Transformers for longer sequences. *ArXiv*, abs/2007.14062.

Xinrong Zhang, Yingfa Chen, Shengding Hu, Zihang Xu, Junhao Chen, Moo Hao, Xu Han, Zhen Thai, Shuo Wang, Zhiyuan Liu, and Maosong Sun. 2024a. ∞Bench: Extending long context evaluation beyond 100K tokens. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15262–15277, Bangkok, Thailand. Association for Computational Linguistics.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2024b. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36.

Yanli Zhao, Andrew Gu, Rohan Varma, Liangchen Luo, Chien chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, and Shen Li. 2023. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16:3848–3860.

Ming Zhong, Da Yin, Tao Yu, Ahmad Zaidi, Mutethia Mutuma, Rahul Jha, Ahmed Hassan Awadallah, Asli Celikyilmaz, Yang Liu, Xipeng Qiu, and Dragomir Radev. 2021. QMSum: A new benchmark for query-based multi-domain meeting summarization. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5905–5921, Online. Association for Computational Linguistics.

A Appendix

A.1 Implementation details

Compatibility with Flash Attention FlashAttention (Dao, 2024) substantially improves the efficiency of standard attention computation. It reduces data movements on GPU by directly producing the output for the attention blocks without storing the $O(L^2)$ attention matrix. However, we rely on these attention scores to select the top K tokens during the full attention steps and construct our partial KV cache C_p (lines 9-10 of Algorithm 2). To make our method compatible with Flash Attention, we implement an extra step to re-compute the attention score at the full attention step. As we do not perform full attention at every generation step, this does not introduce significant overhead. For methods that require accessing attention score (e.g. H_2O), we apply the same procedure to make them compatible with Flash Attention.

Baseline Settings For StreamingLLM, we follow the original paper and maintain a cache with 4 sink tokens and K - 4 recent tokens. For H_2O, we set the heavy hitter size and recent cache size to be $K/2$ each following (Zhang et al., 2024b). For SnapKV, we set the observation window size to 1 and the kernel size to 7 for both RefreshKV and SnapKV following Li et al. (2024). We apply the same aggregation method (max over all query heads) for SnapKV and H_2O for the GQA models.

Continued pretraining We randomly sample a subset of 200k sequences from the Arxiv split of RedPajama dataset[7] and filter out sequences with

[7] https://huggingface.co/datasets/togethercomputer/RedPajama-Data-1T

Dataset	# Example	# In	# Out
RULER	1.3K	32K	<30
QMSum	100	10K	0.1K
GovReport	100	10K	0.7K
NovelSumm	103	100K	1.0K
HTML To TSV (0.5K)	50	18K	0.5K
HTML To TSV (1K)	50	35K	1.6K
HTML To TSV (2K)	50	38K	4.6K
Chain of Keys	100	22K	50

Table 6: Dataset statistics. We report the number of tokens for both the input context and output generation for each dataset, as well as total number of examples.

less than 8192 tokens We train Llama-3.1-8B for one epoch with a global batch size of 64 and a learning rate of 5e-6. We use 20 warm-up steps and a linear schedule with 0 weight decay. We use the AdamW Optimizer. We use Fully Sharded Data Parallel (Zhao et al., 2023) and 8-bit optimizer (Dettmers et al., 2021) to improve training efficiency. Training is done on 4 H100 80 GB GPUs.

A.2 Memory and time requirement comparison

Table 7 compares the memory and attention compute requirements of **RefreshKV** with baselines. We report the memory required to store the KV cache for the L input tokens, and attention compute required to generate the next T tokens.[8] We set our partial cache to be the same size as the complete cache of the eviction-based methods. Under this setting, RefreshKV requires larger KV cache memory compared to eviction-based baselines, but similar to vanilla attention ($L + K$ vs L, where $K << L$). However, our decoding latency is on par with the baselines. Our efficiency depends on two sets of hyperparameters – the partial cache size K, and QC stride and s, which determines how often full attention is performed. By setting $K << L$ and a large S, we can achieve wall clock times similar to KV eviction-based baselines.

A.3 Attention score aggregation for models with GQA

We report language modeling results with different aggregation methods across attention scores of query heads in the same group for models with Grouped Query Attention in Table 8. We see that aggregating over the attention score of the entire group works better than using attention score of one of the head, with taking the max slightly outperforming mean.

A.4 Tuning *s* for query similarity schedule

To choose a similarity threshold s for the dynamic schedule, we run RefreshKV on a held-out set of 50 examples from the Book split of the RedPajama dataset. We evaluate on *QC* stride of {5, 10} with threshold s of {0.80, 0.85, 0.90, 0.95} for Llama-3.1-8B and Qwen2-7B.

Table 9 reports the results of different settings for perplexity and decoding time measured on one A100 machine with batch size of 1. We can see that for Llama-3.1-8B, setting a threshold of 0.85 achieves similar performance for both stride compared to 0.90 and 0.95. In contrast the performance of Qwen2-7B continues to increase going from threshold of 0.80 to 0.95. Therefore, we set the threshold to 0.85 for Llama-3.1-8B and 0.95 for Qwen2-7B.

A.5 Effective stride

We plot the effective stride across layers for Llama-3.1-8B and Qwen2-7B in Figure 4 for the three tasks reported in Table 3. Leveraging query similarity enables dynamic strides across layers for both models. We observe distinct pattern for the two models, with Llama-3.1-8B having a larger stride in the first few layer and Qwen2-7B in the middle layer. We also observe slightly different patterns for different tasks, showing that our method enables flexible scheduling based on the context.

A.6 Chain-of-key task set-up

Task set-up The model is provided with a long lists of keys, each of which contains W number of words, for instance: `apricot-waggish` where $W = 2$. The model is tasked to generate a sequence which consists of a list of T keys from the context, such that the first word of the next key is the last word of the current key. For example: `waggish-fishery`, `fishery-mosquito`, `mosquito-perfume`, `perfume-panda`, `panda-juice` for $T = 5$. We provide an example input in Table 11.

Data generation We first generate a list of English words. We then pair each word with another word to form a list of keys. We ensure that for each key k_1 in the context, there exists exactly one other key k_2 that satisfies the constraint (i.e. the

[8]The KV memory requirements also increases with T. We do not account for this in the table.

	Vanilla	H_2O	StreamingLLM	SnapKV	RefreshKV (Ours)
Memory	L	K	K	K	$L + K$
Time	$T \times L$	$T \times K$	$T \times K$	$T \times K$	$T \times \frac{L}{S} + T \times K$

Table 7: Comparing memory (KV cache size for L input tokens) and time (attention computation for generating the next T tokens) of RefreshKV and baselines. We denote S as stride and use the same KV cache size (K) for the partial cache for our method and complete cache for eviction-based baselines.

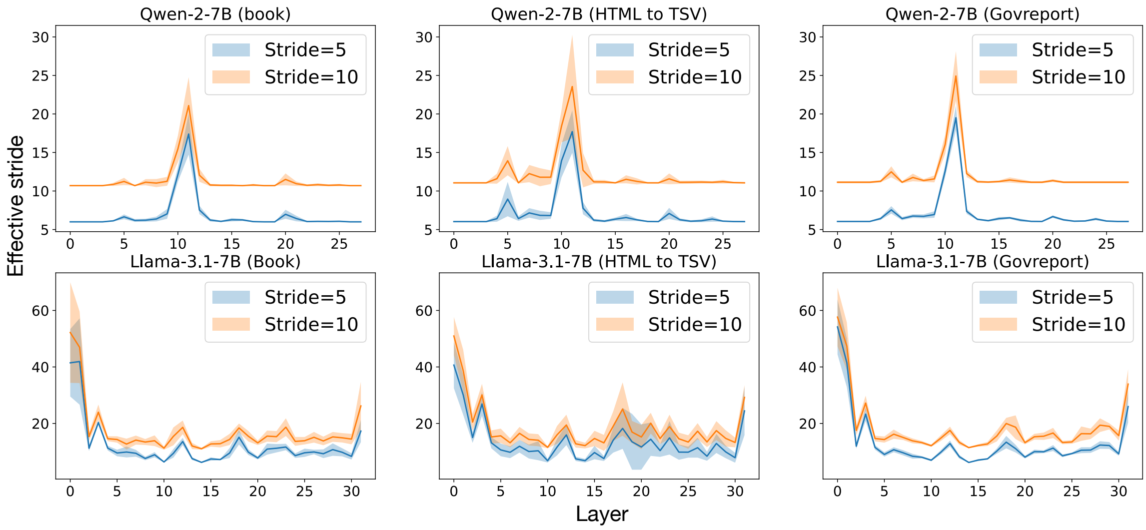

Figure 4: Effective stride across layer for Llama-3.1-8B (similarity threshold=0.85) and Qwen2-7B (similarity trheshold=0.95) in three datasets. We sample 10 examples from each dataset to esimate the effective stride.

Method	**Agg**	**Llama-3.1-8B**	**Qwen-2-7B**
Vanilla	-	2.22/7.07	2.33/8.26
RefreshKV	First	2.34/7.43	2.49/8.78
RefreshKV	Mean	2.32/7.40	2.47/8.73
RefreshKV	Max	2.32/7.40	2.47/8.72

Table 8: Results comparing different methods to aggregate attention scores for GQA models. We experiment with taking the attention score of the first query head, the average and max attention scores of the query heads in the same group to select topK KV cache. For StreamingLLM and RefreshKV, we set $K = 1/8L$ and stride as 10.

first word of k_2 is the last word of k_1). The keys are randomly shuffled in the context.

Evaluation We evaluate correctness of the generated output by the length of a valid chain, divided by T. A valid chain needs to satisfy two criteria: (a) all the key must be in the context and (b) the first word of the current key must be the last word of the previous key. We provide example outputs and their correctness score in Table 12.

Method	**QC stride**	**s**	**Book PPL**	**Time**
		Llama-3.1-8B		
Vanilla	-	-	6.70	7.54
RefreshKV	5	0.80	6.92	6.42
RefreshKV	5	0.85	**6.86**	6.64
RefreshKV	5	0.90	6.88	7.01
RefreshKV	5	0.95	6.88	7.53
RefreshKV	10	0.80	6.95	6.37
RefreshKV	10	0.85	6.96	6.52
RefreshKV	10	0.90	6.96	6.54
RefreshKV	10	0.95	6.95	7.07
		Qwen-2-7B		
Vanilla	-	-	7.44	9.11
RefreshKV	5	0.80	7.86	6.50
RefreshKV	5	0.85	7.80	6.64
RefreshKV	5	0.90	7.73	6.91
RefreshKV	5	0.95	**7.66**	7.14
RefreshKV	10	0.80	7.95	6.37
RefreshKV	10	0.85	7.87	6.41
RefreshKV	10	0.90	7.84	6.62
RefreshKV	10	0.95	7.82	6.67

Table 9: Results of different similarity threshold s on the held-out set of the Book dataset across two QC stride.

Method	Stride	0.5K	2K	8K	Aggregated
		Llama-3.1-8B			
Vanilla		43	31	23	33
Streaming		4	1	0	2
SnapKV		0	0	0	0
H2O		0	0	0	0
Refresh	QC=5	**31**	**15**	**4**	**17**
Refresh	QC=10	16	7	1	8
		Qwen-2-7B			
Vanilla		36	22	15	24
Streaming		10	**3**	0	5
SnapKV		0	0	0	0
H2O		0	0	0	0
Refresh	QC=5	**20**	**6**	**3**	**10**
Refresh	QC=10	14	2	1	6

Table 10: Breakdown of HTML tasks based on output length.

A.7 Results on LongProc tasks with short inputs

Task set-up We report results on 4 more tasks from **LongProc** (Ye et al., 2025): **Path Traversal**, **Travel Planning**, **Countdown** and **Theory-of-mind tracking**. These tasks consist of input with less than 10K tokens. While **Path Traversal** consists of a version with 12K input tokens, we exclude it from our main results as none of the open sourced models are able to perform the task in vanilla setting. We report results on 50 samples for each task. We set $K = 1/8L$ for RefreshKV and baselines.

Evaluation We follow evaluation practice of the original paper (Ye et al., 2025). For Countdown and Travel Planning, we report correctness of the final solution using rule-based validators. For Path Traversal and ToM Tracking, we report accuracy.

Results Results of RefreshKV and baseline methods are in Table 13. We observe similar trend as the **HTML to TSV** task – Most of the baselines fail completely on the task. RefreshKVwith $QC = 5$ recovers 50% and 60% performance of full attention for Llama-3.1-8B and Qwen2-7B respectively.

A.8 Detailed RULER results

We follow the suite of evaluation tasks introduced in (Hsieh et al., 2024), which consists of the 13 tasks.[9] We refer the readers to Hsieh et al. (2024) for detailed description and examples of each task and Appendix B for the exact tasks configurations. We group them based on the types:

- **Single NIAH** An NIAH-styled task with one key and one value to retrieve. We include three variations of the task with different types of key, value and haystack.
- **Multi-key NIAH** An NIAH-styled task with distracting keys. We include three variations of the task with different types of key, value and haystack.
- **Multi-value NIAH** An NIAH-styled task with multiple values corresponding to the key.
- **Multi-query NIAH** An NIAH-styled task with multiple queries, each corresponding to a distinct key.
- **Variable Tracking** A NIAH-styled task that requires tracing through multiple hops.
- **Common word extraction** and **Frequent word extraction** require extracting the words based on the pattern in a list of words. Common word extraction expects a list of 10 most common words while frequent word extractions expect a list of 3 frequent words.
- **Question Answering** A task that requires answering a question given a set of documents. We include two variations of the tasks, corresponding to two question answering datasets.

Per-task results We report detailed performance of RULER subtasks in Table 14, grouped by task type. For both models, the best baselines (SnapKV) achieves comparable results as RefreshKV for tasks with short-form outputs, such as **Single NIAH**. However, for tasks that require longer outputs, such as **Multi-key** and **Multi-value NIAH**, RefreshKV outperform all the baselines.

[9] https://github.com/hsiehjackson/RULER

Input
"You are given many keys composed of a few words. Your task is to generate a chain of 10 keys such that the first word of the current key is the last word of the previous key. Separate the keys with comma. Example: waggish-fishery, fishery-mosquito, mosquito-perfume, perfume-panda, panda-juice, juice-willow, willow-bronco, bronco-creditor, creditor-bathhouse, bathhouse-woman. You must generate keys that are in the context. DO NOT REPEAT THE EXAMPLE. Context:Name of key: toga-roommate Name of key: appetiser-cenario Name of key: normalization-tacit Name of key: intensity-ping Name of key: innate-cummerbund Name of key: tentacle-lining [...omitted...] Name of key: breath-yielding Name of key: schema-festive You are given many keys composed of a few words. Your task is to generate a chain of 10 keys such that the first word of the current key is the last word of the previous key. Separate the keys with comma.You must generate keys that are in the context. Chain of ten keys:"

Table 11: Example input for the chain-of-key task where $W = 2$ and $T = 10$.

Output	**Score**
`impossible-crawdad, crawdad-vehicle, vehicle-uncertainty, uncertainty-welfare, welfare-outrigger, outrigger-historical, historical-gator, gator-hugger, hugger-debris, debris-precious`	1 (fully correct)
`annoying-pentagon, pentagon-fit, fit-waggish, waggish-fishery, fishery-mosquito, mosquito-perfume, perfume-panda, panda-juice, juice-willow, willow-bronco`	0.2 (correct up to the second key)
`impossible-crawdad, crawdad-vehicle, vehicle-uncertainty, welfare-outrigger, outrigger-historical, historical-gator, gator-hugger, hugger-debris, debris-precious, uncertainty-welfare`	0.3 (correct up to the third key)

Table 12: Example output for the chain-of-key task where $W = 2$ and $T = 10$ and their score. Keys that are not in the context are highlighted in red.

Method	**stride**	**Path Traversal**	**ToM Tracking**	**Countdown**	**Travel Planning**
		Llama-3.1-8B			
Vanilla	-	17	40	67	62
StreamingLLM	-	0	0	0	0
H_2O	-	0	0	0	2
SnapKV	-	1	0	12	0
RefreshKV	QC=5	**5**	**14**	**44**	**38**
RefreshKV	QC=10	1	5	42	18
		Qwen-2-7B			
Vanilla	-	7	12	11	48
StreamingLLM	-	2	0	6	0
H_2O	-	2	0	6	0
SnapKV	-	0	0	**14**	2
RefreshKV	QC=5	**3**	**6**	**14**	**26**
RefreshKV	QC=10	2	2	10	4

Table 13: Performance on long-context tasks with short outputs from LongProc benchmark for LLaMA-3.1-8B-Instruct and Qwen-2-7B-Instruct.

Method	niah_single	multi_key	multi_query	multi_value	fwe	vt	cwe	qa
			Llama-3.1-8B					
Vanilla	100	98	99	99	93	99	65	61
H_2O	7	7	6	6	78	38	39	34
Streaming	8	13	13	13	93	12	4	42
SnapKV	99	60	**98**	**99**	**83**	**99**	**44**	**63**
RefreshKV(QC=5)	**100**	**91**	**98**	**99**	81	**99**	**44**	60
RefreshKV(QC=10)	**100**	67	97	**99**	81	**99**	**44**	59
			Qwen-2-7B					
Vanilla	100	90	75	87	84	86	27	50
H_2O	5	8	5	3	**84**	2	17	30
Streaming	8	11	13	12	80	15	14	39
SnapKV	69	51	54	43	81	**87**	**27**	**50**
RefreshKV(QC=5)	**99**	**79**	**70**	**85**	70	**87**	**27**	**50**
RefreshKV(QC=10)	97	54	63	67	80	87	**27**	49

Table 14: Detailed performance of RULER subtasks with $L = 32K$. For non-vanilla methods, we set the $K = 1/8L$.