Efficient KV Cache Management for Long-Context LLM Inference: Lazy Pruning, Slack, and Staged Eviction

Anonymous Author(s)
Affiliation
Address
email

Abstract

Long-context LLM inference is bottlenecked by KV-cache growth and per-token runtime overhead. StreamingLLM [10] bounds KV-cache size by retaining attention sinks and a recent-token window, but practical decoding can suffer from (i) non-trivial KV pruning overhead and (ii) quality degradation caused by periodic hard evictions, especially on long-form generation. We propose three lightweight, training-free KV-cache management mechanisms on top of StreamingLLM: **Lazy Pruning** (amortize pruning by triggering after R tokens of overflow), **Slack** (a hard-cap buffer that enables staged evictions under bounded memory), and **Max_Drop** (limit the maximum tokens evicted per prune event). We evaluate on Pythia-2.8B using WikiText-103 (sanity) and PG19 (long-context), and report TTFT/TPOT/throughput/memory/perplexity trade-offs. We also document why FlashAttention, speculative decoding, quantization, and CUDA operator fusion did not improve batch-1 streaming decode under our setting.

1 Introduction

Long-context LLM inference is challenging because both computation and memory scale with the context length. During autoregressive decoding, the KV cache grows linearly with the number of processed tokens, increasing attention cost and memory traffic. StreamingLLM [10] mitigates this by keeping only (i) the first S "sink" tokens and (ii) a sliding window of the most recent W tokens, thereby bounding the effective KV-cache length.

Motivation. In our setting (Pythia-2.8B on NVIDIA A800), once attention length is bounded, decode becomes dominated by MLP and framework/launch overhead. Meanwhile, KV pruning itself can incur additional overhead due to slicing/copy and rotary-position re-alignment. More importantly, periodic hard eviction introduces distribution shifts that manifest as token-level NLL spikes and perplexity (PPL) degradation on long-form datasets. These observations motivate KV-management mechanisms that (i) reduce amortized pruning overhead and (ii) stage evictions to stabilize quality, without modifying attention kernels or model weights.

Contributions. We propose three training-free, modular mechanisms on top of StreamingLLM: (i) **Lazy Pruning**, which triggers pruning only after the cache overflow reaches R tokens; (ii) **Slack**, a hard capacity buffer of σ tokens that supports staged pruning; (iii) **Max_Drop**, which limits the maximum tokens evicted per prune event to δ. We provide a reproducible evaluation pipeline and summarize negative results for methods that are effective in other regimes but do not improve batch-1 streaming decode.

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

2 Related Work

Long-context inference and KV-cache compression. Test-time KV-cache management is a widely used approach to make long-context inference feasible, including retention/eviction heuristics and cache compression [10, 11, 8, 6]. Our work builds on StreamingLLM's Start+Recent rule and focuses on *training-free* extensions that target the practical costs of pruning and eviction-induced distribution shifts. **Kernel- and decoding-level accelerations.** Attention kernels such as FlashAttention aim to reduce attention IO cost [3, 2], while speculative decoding accelerates generation by drafting tokens [7, 1]. Post-training quantization reduces compute/memory but its benefits depend on hardware and runtime regime [4, 5]. We empirically find these do not improve batch-1 streaming decode in our setting and analyze the underlying reasons (Sec. 5).

3 Method

We first define Start+Recent streaming, then formalize our three mechanisms.

3.1 Preliminaries: Start+Recent StreamingLLM

Let $x_{1:t}$ be the processed prefix at decode step t. For each layer ℓ, let $(\mathbf{K}_t^\ell, \mathbf{V}_t^\ell)$ denote the KV cache with sequence length L_t along the cache dimension. Start+Recent streaming retains (i) the first S sink tokens, and (ii) the most recent W tokens. Define the soft capacity

$$C_0 \triangleq S + W. \tag{1}$$

Our code also supports optional *non-core* retained tokens (e.g., overlap/refresh); we denote their total budget by $B \geq 0$ and define the effective soft capacity

$$C \triangleq C_0 + B. \tag{2}$$

Unless stated otherwise, we use $B = 0$ in paper experiments to isolate our three mechanisms. When pruning is performed, the retained index set is

$$\mathcal{I}_t \;=\; \{0, 1, \dots, S-1\} \;\cup\; \{L_t - W, \dots, L_t - 1\}, \tag{3}$$

with the recent segment clamped to avoid overlap with the sink segment.

RoPE consistency. For rotary-position-embedding models (e.g., Pythia/GPT-NeoX), pruning changes token positions in the cache; therefore, cached keys must be re-aligned to the new positions. Let $p \in \mathcal{I}_t$ be an old token position and $p' = f_t(p) \in \{0, \dots, |\mathcal{I}_t| - 1\}$ be its new position after compaction. RoPE encodes position via per-dimension frequencies ω_i (from `inv_freq`). Re-alignment is equivalent to applying a *delta-rotation* with $\Delta p = p' - p$:

$$\begin{bmatrix} k'_{2i} \\ k'_{2i+1} \end{bmatrix} \;=\; \begin{bmatrix} \cos(\omega_i \Delta p) & -\sin(\omega_i \Delta p) \\ \sin(\omega_i \Delta p) & \cos(\omega_i \Delta p) \end{bmatrix} \begin{bmatrix} k_{2i} \\ k_{2i+1} \end{bmatrix}. \tag{4}$$

Our implementation exploits that Start+Recent yields a constant Δp for the contiguous recent block (a constant shift), and falls back to the general per-token re-rotation when tokens are explicitly relocated (e.g., refresh tokens).

3.2 Lazy Pruning

Naïve streaming prunes as soon as the cache exceeds C, which can add overhead due to slicing/copy and KV relocation. We instead amortize pruning by allowing the cache to exceed C by up to $R-1$ tokens before triggering a prune. Define overflow:

$$\mathrm{overflow}_t \triangleq L_t - C. \tag{5}$$

Let R be the *overflow allowance* hyperparameter (in code: `compress_every`). We allow $R = 0$ to disable pruning for debugging (unbounded memory), and use $R \geq 1$ in all bounded-memory settings. Lazy pruning triggers only when overflow reaches R:

$$\mathrm{PruneTrigger}(t) \triangleq \mathbb{1}[R > 0] \cdot \mathbb{1}\big[\mathrm{overflow}_t \geq R\big]. \tag{6}$$

This matches the implementation semantics and avoids the ambiguity that would arise from interpreting $R = 0$ as "prune immediately".

Algorithm 1 KV pruning with Lazy Pruning, Slack, and Max_Drop

Require: sink size S, window size W, extra budget B (default 0), overflow allowance R (0 disables), slack σ, max_drop δ

$C_0 \leftarrow S + W$, $C \leftarrow C_0 + B$, $H \leftarrow C + \sigma$
observe cache length L_t
if $R = 0$ **then**
    return
end if
if $L_t \leq C$ **then**
    return
end if
if $L_t - C < R$ **then**
    return
end if
if $\delta = 0$ **then**
    $L_t^\star \leftarrow C$
else
    $L_t^\star \leftarrow \min(\max(L_t - \delta,\ C),\ H)$
end if
keep first S tokens and most recent $(L_t^\star - S)$ tokens
prune KV cache and re-align RoPE positions

3.3 Slack (Hard-Cap Buffer)

Slack introduces a *hard* capacity that bounds temporary cache growth during staged pruning. Specifically, we define

$$H \triangleq C + \sigma, \tag{7}$$

where $\sigma \geq 0$ is a fixed slack budget. In our implementation, σ does *not* change the pruning trigger; it caps the post-prune target length when combined with Max_Drop (Sec. 3.4).

3.4 Max_Drop (Staged Eviction)

Periodic pruning can cause "eviction cliffs" when a large block is removed at once, which we observed as token-level NLL spikes on long-form generation. Max_Drop limits the maximum number of tokens evicted in a single prune event.

Let $\delta \geq 0$ be the maximum drop size. When pruning is triggered, we set a target retained length

$$L_t^\star = \begin{cases} C, & \delta = 0 \\ \min\big(\max(L_t - \delta,\ C),\ H\big), & \delta > 0 \end{cases} \tag{8}$$

and prune the cache to keep exactly $L_t^\star$ tokens using the Start+Recent rule. This turns a single large eviction into multiple smaller evictions across steps while remaining within the hard capacity H.

Hyperparameter interactions and a toy example. When $\delta > 0$, staged eviction can leave $L_t^\star > C$, so pruning may trigger again soon; this is intentional. A practical rule-of-thumb to avoid pruning *every* step when the cache saturates at H is to choose $\sigma < R$ (so that after pruning to H, the overflow remains $< R$ until enough new tokens arrive). For example, with $(S, W, R, \sigma, \delta) = (4, 2044, 32, 16, 32)$ (so $C = 2048$, $H = 2064$), if a step reaches $L_t = 2090$ (overflow $= 42$), we prune to $L_t^\star = \min(\max(2090 - 32, 2048), 2064) = 2058$ and continue staging further evictions as decoding proceeds.

3.5 Algorithm Summary

Algorithm 1 summarizes the pruning control logic (RoPE re-alignment and buffer management are omitted for clarity).

Table 1: Main results on PG19 (long-context, *fill values via script output*).

Method	TPOT↓	Speedup↑	PPL↓	Peak Mem (MB)↓
Baseline (Full KV)	100.53	1.00×	19.761	5722
StreamingLLM (MIT)	[INSERT DATA]	[INSERT DATA]	[INSERT DATA]	[INSERT DATA]
Ours (Lazy/Slack/Max_Drop)	[INSERT DATA]	[INSERT DATA]	[INSERT DATA]	[INSERT DATA]

Table 2: Sanity-check results on WikiText-103 (short-context).

Method	TPOT↓	Speedup↑	PPL↓
Baseline (Full KV)	98.86	1.00×	9.359
StreamingLLM (MIT)	[INSERT DATA]	[INSERT DATA]	[INSERT DATA]
Ours (Lazy/Slack/Max_Drop)	[INSERT DATA]	[INSERT DATA]	[INSERT DATA]

4 Experiments

4.1 Setup

We evaluate on Pythia-2.8B. We use WikiText-103 as a short-context sanity check and PG19 as the primary long-context benchmark [9]. We report TTFT, TPOT (ms/token), throughput (tok/s), total runtime, peak GPU memory, and perplexity (PPL). **Baselines and fairness.** Our long-context evaluation must respect the model's maximum position length. Therefore, we compare methods under the same *effective context cap* C (Sec. 3): the non-streaming baseline uses a sliding window of length C and recomputes attention each step (no KV cache), while StreamingLLM variants reuse KV cache and apply pruning/re-alignment to stay within the same cap. All methods share the same model, dataset segment, and evaluation protocol implemented in `experiments/eval_streaming_llm.py`.

Protocol and repeatability. To reduce measurement noise, our reproduction script runs a warmup pass followed by N repeated trials for each configuration and reports mean and standard deviation (saved in JSON). Each JSON record includes environment fingerprints (GPU, driver, torch/CUDA) to prevent reusing baselines across incompatible environments.

4.2 Main Results and Ablations

We provide a one-click script (`run_paper_experiments.sh`) that runs all configurations and generates LaTeX tables automatically. If the tables are not generated yet, we show placeholders.

Implementation-path fairness check. To ensure that improvements are not caused by divergent code paths, we also report an *Ours-framework-only* configuration that uses our wrapper implementation but matches Start+Recent semantics (MIT-style cache backend) with all proposed mechanisms disabled.

Ablations.

5 Discussion

Why do popular inference accelerations fail in batch-1 streaming? In our setting, streaming bounds attention length, shifting the bottleneck toward MLP and framework/launch overhead. Therefore, further optimizing attention kernels can have limited impact. Moreover, methods that assume static shapes or stable cache semantics may be incompatible with pruning and RoPE re-alignment in streaming decode.

Attempts and analysis of ineffective methods. We tested several common acceleration routes (see our project log in `docs/`): (i) FlashAttention integration was non-trivial under RoPE re-alignment and did not improve batch-1 TPOT in our environment; (ii) speculative decoding was unreliable on long-form generation and incompatible with streaming cache management; (iii) quantization via TorchAO exhibited numerical issues (INT8 v1) and remained slower even when stabilized

Table 3: Ablation study for Slack and Max_Drop (PG19).

Setting	TPOT↓	Speedup↑	PPL↓
w/o Slack ($\sigma = 0$)	[INSERT DATA]	[INSERT DATA]	[INSERT DATA]
w/o Max_Drop ($\delta = 0$)	[INSERT DATA]	[INSERT DATA]	[INSERT DATA]
Full (Lazy+Slack+Max_Drop)	[INSERT DATA]	[INSERT DATA]	[INSERT DATA]

Table 4: Summary of investigated but ineffective optimization routes in our batch-1 streaming setting.

Method	Outcome	Notes (see project logs)
FlashAttention / FlashDecoding	No gain / hard to integrate	Streaming already bounds attention length; remaining bottlenecks are MLP and launch/framework overhead; RoPE re-alignment complicates clean integration.
Speculative decoding	No gain / unreliable	Long-form generation yields low acceptance; cache consistency with pruning is fragile.
Quantization (TorchAO INT8/INT4)	Slower / unstable	INT8 WO v1 produced NaNs; v2 is stable but slower for batch-1 decode in our stack; INT4 backend dependencies were problematic.
`torch.compile` / CUDA Graphs	Unstable	Repeated-run CUDA graph overwrite errors observed in rotary-embedding path; shape/cache semantics hinder capture.
HF StaticCache	Incompatible	StaticCache assumes fixed cache updates; pruning can trigger device-side asserts (index out of bounds).
CUDA fusion (residual/LN)	No gain	Amdahl's law: residual/LN is a small fraction; custom kernel launch overhead dominated, leading to slowdown.

(INT8 v2) in batch-1 streaming decode; (iv) `torch.compile` with CUDA Graphs was unstable due to overwritten outputs in rotary-embedding paths; (v) HuggingFace StaticCache conflicted with pruning updates and triggered device-side asserts. These results highlight that strong batch-inference accelerations do not directly transfer to batch-1 streaming decode.

6 Conclusion

We presented three modular KV-cache management mechanisms for StreamingLLM: Lazy Pruning, Slack, and Max_Drop. Our method is training-free and targets long-context decoding where pruning overhead and eviction-induced quality loss become important. We provide a reproducible pipeline and report negative results to clarify which acceleration techniques do or do not apply in this regime.

References

[1] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[2] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[3] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *NeurIPS*, 2022.

[4] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. *NeurIPS*, 2022.

[5] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. In *ICLR*, 2023.

[6] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *SOSP*, 2023.

[7] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *ICML*, 2023.

[8] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *arXiv preprint arXiv:2305.17118*, 2023.

[9] Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, Chloe Hillier, and Timothy P Lillicrap. Pg-19 language modeling benchmark. arXiv preprint arXiv:1911.05507, 2019.

[10] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *ICLR*, 2024.

[11] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. In *NeurIPS*, 2023.