Efficient KV Cache Management for Long-Context LLM Inference: Lazy Pruning for Practical StreamingLLM

Jifan Lin (523030910057)
Shanghai Jiao Tong University
ljf2570@sjtu.edu.cn

Second Author Name (XXXXXXXXXXXXX)
Shanghai Jiao Tong University
email2@sjtu.edu.cn

Abstract

Long-context LLM inference is bottlenecked by KV-cache growth and per-token runtime overhead. StreamingLLM Xiao et al. [2024] bounds KV-cache size by retaining attention sinks and a recent-token window, but practical decoding can still suffer from *non-trivial cache-management overhead*, especially for RoPE models where pruning changes cache positions and requires positional consistency. We implement a modular StreamingLLM stack on HuggingFace (Cache API) and propose **Lazy Pruning**: a training-free strategy that batches Start+Recent compaction and RoPE re-alignment, amortizing expensive cache slicing/copy/rotation across multiple decode steps. On PG19 (20k tokens, Auto-cap 2048), Lazy Pruning improves TPOT from 19.57ms (strict Start+Recent) to 14.37ms and achieves 6.81$\times$ speedup over a sliding-window recompute baseline; its PPL increases from 19.761 to 20.318 (+2.82% vs recompute, +0.68% vs strict Start+Recent). Additional negative results are summarized in Appendix.

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 Xiao et al. [2024] 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 a KV-management mechanism that reduces amortized pruning overhead without modifying attention kernels or model weights.

Contributions. We make three contributions centered on a *practical* StreamingLLM implementation and an amortization mechanism: (i) we implement StreamingLLM on top of HuggingFace's Cache API while keeping attention modules unmodified, and explicitly handle RoPE consistency as a cache-side operation for plug-and-play comparisons; (ii) we propose **Lazy Pruning**, which batches Start+Recent compaction and RoPE re-alignment to reduce amortized cache-management overhead in batch-1 decode; (iii) we provide a reproducible pipeline and a detailed exploration log of negative results, clarifying which optimizations do or do not transfer to batch-1 streaming decode.

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 Xiao et al. [2024], Zhang et al. [2023], Liu et al. [2023], Kwon et al. [2023]. 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 Dao et al. [2022], Dao [2023], while speculative decoding accelerates generation by drafting tokens Leviathan et al. [2023], Chen et al. [2023]. Post-training quantization reduces compute/memory but its benefits depend on hardware and runtime regime Dettmers et al. [2022], Frantar et al. [2023]. 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 and our cache-side RoPE re-alignment, then formalize Lazy Pruning. Additional exploratory mechanisms are summarized in Appendix.

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 Lazy Pruning.

Auto-cap (fixed total context budget). To avoid confounds from changing the *total* number of visible tokens, and to respect the model's maximum context length, we adopt an **Auto-cap** protocol in our evaluation. We fix a global cap C_{cap} (e.g., 2048 for Pythia-2.8B) and count all components that increase the visible KV length into the same budget:

$$S + W + \sigma + O + B \ \leq \ C_{\text{cap}}, \tag{3}$$

where σ is an optional slack budget (used only in exploratory experiments), O is overlap (if enabled), and B is the refresh budget. Equivalently, for a fixed C_{cap} we derive the recent-window size as

$$W \ = \ C_{\text{cap}} - S - \sigma - O - B. \tag{4}$$

This defines the post-prune *target length* and prevents methods from silently seeing more tokens when enabling slack/overlap/refresh, making speed–quality comparisons reproducible. Under strict Start+Recent pruning (R=1), the cache is kept at C_{cap} every step. Under Lazy Pruning with interval R, the cache is still pruned back to C_{cap}, but can temporarily overflow between prunes; in our implementation the peak cache length during a forward pass is bounded by $C_{\text{cap}} + R$ (Fig. 1).

Peak-length bound (fairness). Because pruning is applied *after* each forward pass, the effective attention length during forward can exceed the post-prune target. For a given pruning interval R, the forward-time cache length satisfies:

$$L_t^{\text{fwd}} \leq C_{\text{cap}} + R. \tag{5}$$

We report speed and quality under a fixed post-prune target C_{cap}, and provide an overhead breakdown (Appendix A2) to attribute speedups to amortized cache-management overhead rather than attention-length effects. When pruning is performed, the retained index set is

$$\mathcal{I}_t \ = \ \{0, 1, \ldots, S-1\} \ \cup \ \{L_t - W, \ldots, L_t - 1\}, \tag{6}$$

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

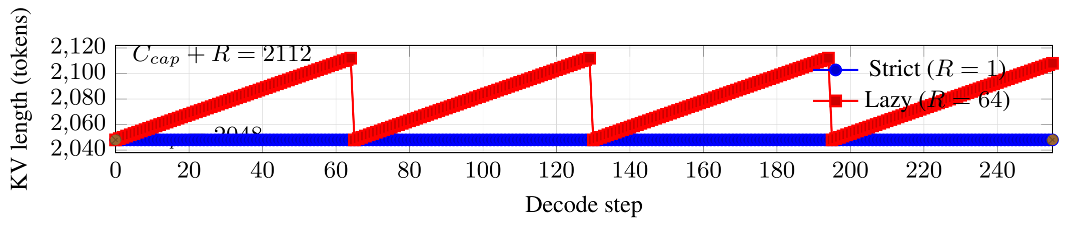

Figure 1: KV-cache length during decoding under Auto-cap. Strict Start+Recent (R=1) prunes every step and stays at C_{cap}. Lazy Pruning (R=64) reduces prune frequency at the cost of bounded oscillation; the peak forward-time length is bounded by $C_{\text{cap}} + R$. KV length is proportional to KV memory footprint and influences the amount of cache compaction work.

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. We implement RoPE consistency as a cache-side operation (without modifying attention forwards); the derivation and implementation details are deferred to Appendix A.3.

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 triggering compaction only when overflow reaches a threshold R (in code: `compress_every`), reducing the frequency of expensive prune+RoPE re-alignment operations. Define overflow:

$$\text{overflow}_t \triangleq L_t - C. \tag{7}$$

Let R be the *pruning interval/allowance* hyperparameter (in code: `compress_every`). In bounded-memory evaluation we use $R \geq 1$. Lazy pruning triggers only when overflow reaches R:

$$\text{PruneTrigger}(t) \triangleq \mathbb{1}\big[\text{overflow}_t \geq R\big]. \tag{8}$$

This choice explicitly trades a slightly larger instantaneous cache length for fewer expensive prune-and-realign events, which is effective when pruning overhead is non-negligible under batch-1 decode. Fig. 1 illustrates the resulting sawtooth behavior. For debugging, our code also supports disabling pruning via `compress_every=0`; we do not use this mode in bounded-memory comparisons.

Rule form. Strict pruning (Start+Recent) prunes whenever $\text{overflow}_t > 0$. Lazy Pruning prunes only when $\text{overflow}_t \geq R$, then compacts back to C and re-aligns RoPE (Appendix A.3).

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 Rae et al. [2019]. Due to space, the main paper focuses on TPOT, speedup over recompute, PPL, and peak GPU memory; additional metrics are logged by our evaluator. **Datasets.** We use WikiText-103 Merity et al. [2016] and PG19 Rae et al. [2019]. **Baselines and fairness.** We compare methods under the same fixed soft cap C_{cap} (Auto-cap; Sec. 3). The non-streaming baseline uses a sliding window of length C_{cap} and recomputes attention each step (no KV cache), while StreamingLLM variants reuse KV cache and apply pruning/re-alignment targeting a post-prune length of C_{cap}; Lazy Pruning may temporarily overflow between prunes but remains bounded. All methods share the same model, dataset segment, and evaluation protocol implemented in `experiments/eval_streaming_llm.py`.

Protocol. We use a fixed evaluation protocol implemented in `experiments/eval_streaming_llm.py` and report TPOT, speedup over recompute, PPL, and peak GPU memory.

Table 1: Main results on PG19 (long-context) under Auto-cap C_{cap}=2048.

Method	TPOT↓	Speedup↑	PPL↓	Peak Mem (MB)↓
Baseline (Sliding Window, no KV)	97.91	1.00×	19.761	5723
StreamingLLM (Start+Recent; strict prune)	19.57	5.00×	20.181	6941
Ours (Lazy Pruning; R=64)	14.37	6.81×	20.318	6941

4.2 Main Results and Ablations

Comparison protocol. We compare a strict Start+Recent baseline (immediate pruning, R=1) against our Lazy Pruning variant (batched pruning, R>1) under a fixed KV budget (Auto-cap, $S+W$=2048). This isolates the effect of pruning frequency and cache-management overhead under the same bounded-attention regime.

Main finding. On PG19 under the same Auto-cap budget, Lazy Pruning reduces TPOT from 19.57ms (strict Start+Recent, R=1) to 14.37ms (R=64), a further 1.36× improvement over strict pruning, with a small additional PPL increase (20.181 → 20.318) and unchanged peak memory in our evaluator. **Evidence for amortization.** Under a controlled PG19 probe where only R changes, the forward-time component stays nearly constant while cache-update overhead drops sharply as prune events become infrequent, consistent with a simple decomposition $\text{TPOT} \approx T_{\text{fwd}} + T_{\text{update}}/R$ (Appendix A2).

Fairness and measurement notes. Auto-cap fixes the post-prune target length to C_{cap}=2048, while Lazy Pruning allows bounded overflow between prunes with peak forward-time length $C_{\text{cap}}+R$. Peak GPU memory is measured as `torch.cuda.max_memory_allocated()` in our evaluator; the additional KV introduced by R=64 is small relative to total memory and may not change the rounded peak value.

Ablations. We report the full ablation ladder, additional R settings, and exploratory quality heuristics (Slack/Max_Drop, overlap/refresh) in Appendix A.4. In the main body we focus on the strict (R=1) vs lazy (R=64) contrast as the primary effect.

5 Discussion

Bottleneck shift: forward dominates after cache management is amortized. After StreamingLLM bounds the attention length and Lazy Pruning amortizes prune+RoPE re-alignment overhead, most remaining runtime concentrates in the model forward pass. We evaluated several standard accelerations (FlashAttention, speculative decoding, quantization, `torch.compile`/CUDA Graphs, StaticCache, and a CUDA micro-kernel), but none provided consistent net gains in our batch-1 long-context streaming regime (Appendix A.5).

6 Conclusion

We present a practical StreamingLLM implementation on HuggingFace and a single effective optimization: Lazy Pruning, which amortizes Start+Recent compaction and RoPE re-alignment overhead in batch-1 streaming decode. On PG19, Lazy Pruning achieves 6.81× speedup over a sliding-window recompute baseline (vs 5.00× for strict Start+Recent) under a fixed context budget with minor PPL degradation (+2.82% vs recompute, +0.68% vs strict).

A Appendix

A.1 Limitations and future work

Our results suggest that in a bounded-attention, batch-1 decoding regime, end-to-end latency quickly becomes forward dominated once KV length is capped and cache-update costs are amortized. In this regime, Lazy Pruning yields a measurable TPOT reduction with only a small additional PPL increase under a fixed post-prune target length. However, our approach has several limitations. (i) The remaining speed bottleneck is the model forward pass (GEMMs and associated pointwise/LN ops); further gains likely require kernel- or graph-level optimization beyond cache management. (ii) Lazy Pruning permits bounded overflow ($C_{\text{cap}}+R$ peak forward-time length), and our peak-memory reporting uses `torch.cuda.max_memory_allocated()`, which can be insensitive to small KV differences at MB granularity; repeated runs and finer-grained reporting (allocated vs reserved; mean$\pm$std) are needed. (iii) We observe a small PPL increase relative to strict Start+Recent and did not find consistently beneficial quality-oriented heuristics under Auto-cap, indicating that improving quality without regressing speed may require different cache-consistency mechanisms. Future work includes: (i) cache layouts with static/ring-buffer semantics to better support compilation/graph capture; (ii) fused transformer-block implementations or alternative backends to reduce forward overhead; and (iii) broader evaluation across models, longer contexts, and multiple runs to quantify variance and robustness.

A.2 Implementation notes and pseudocode

We summarize the core pruning rule (Start+Recent) and our Lazy Pruning trigger (R) in Algorithm 1. This pseudocode matches the implementation setting used in our evaluator (Auto-cap, prune-to-C_{cap}).

Algorithm 1 KV pruning with Lazy Pruning (implementation-level summary)

```
Require: sink size $S$, window size $W$, extra budget $B$ (default 0), pruning interval $R \geq 1$
 1: $C \leftarrow S + W + B$
 2: observe cache length $L_t$
 3: if $L_t \leq C$ then
 4:     return
 5: end if
 6: if $L_t - C < R$ then
 7:     return
 8: end if
 9: keep first $S$ tokens and most recent $(C - S)$ tokens
10: prune KV cache and re-align RoPE positions
```

A.3 RoPE consistency derivation

For rotary-position-embedding models, pruning changes token positions in the cache; therefore cached keys must be re-aligned to the new positions. Let p be an old token position and p' its new position after compaction. RoPE encodes position via per-dimension frequencies ω_i (from `inv_freq`). Re-alignment is equivalent to a delta rotation with $\Delta p = p' - p$:

$$\begin{bmatrix} k'_{2i} \\ k'_{2i+1} \end{bmatrix} = \begin{bmatrix} \cos(\omega_i \Delta p) & -\sin(\omega_i \Delta p) \\ \sin(\omega_i \Delta p) & \cos(\omega_i \Delta p) \end{bmatrix} \begin{bmatrix} k_{2i} \\ k_{2i+1} \end{bmatrix}. \tag{9}$$

In the Start+Recent setting, the recent block undergoes a constant position shift, enabling reuse of rotation factors across layers.

A.4 Full experimental results

Table A1: Sanity-check results on WikiText-103 (short-context).

Method	TPOT↓	Speedup↑	PPL↓
Baseline (Sliding Window, no KV)	97.36	1.00×	9.359
StreamingLLM (Start+Recent; strict prune)	20.22	4.82×	9.496
Ours (Lazy Pruning)	14.45	6.74×	9.519

Table A2: Profiling evidence for amortization (PG19 probe; fixed $C_{\text{cap}} = 2048$, $S = 32$, $\sigma = 16$): forward time stays nearly constant while cache-update overhead is amortized by increasing R.

Setting	Prune events	Forward (ms)	Update (ms)	Total (ms)
Strict ($R = 1$)	2000	14.12	5.57	19.69
Lazy ($R = 64$)	31	14.03	0.09	14.12

Table A3: Ablation ladder on PG19 (aligned S, W). Differences below ~1% may fall within run-to-run noise unless stated otherwise.

Setting	TPOT↓	Speedup↑	PPL↓
Start+Recent (strict prune) (R=1)	19.57	5.00×	20.181
Start+Recent (strict prune; framework-only) (R=1)	20.02	4.89×	20.181
+ Lazy Pruning (R=64)	14.38	6.80×	20.318
+ (expl.) Slack (R=64, σ=16)	14.35	6.82×	20.262
+ (expl.) Slack + Max_Drop (R=64, σ=16, δ=32)	14.53	6.74×	20.272

A.5 Negative results (evidence-backed)

Table A4: Summary of investigated but ineffective optimization routes in our batch-1 streaming setting.

Method	Outcome	Notes (see project logs)
FlashAttention / FlashDecoding	Inconclusive / low ROI	Integration is sensitive to environment and attention implementation; once streaming bounds attention length, speed is often dominated by MLP and launch/framework overhead.
Speculative decoding	Negative result	In long-context workloads, draft/target mismatch yields low acceptance rates, making SpecDec slower than normal decoding.
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)	Slower	Amdahl's law: residual/LN is a small fraction; custom kernel launch overhead dominated (0.92×, TPOT +8.6%).

Speculative decoding in long-context workloads. We evaluated speculative decoding with small/medium draft models in a long-context PG19 workload. While a *draft=target* sanity check achieved near-perfect acceptance, using smaller drafts led to low acceptance rates and poor tokens-per-target-forward, making speculative decoding slower than normal decoding. We attribute this to draft/target distribution mismatch at long positions.

FlashAttention integration. We prepared an evaluation path that can switch attention backends (math/SDPA/FlashAttention when available). In our regime, streaming already bounds attention length, so overall speed is often dominated by MLP and framework/launch overhead; additionally, FlashAttention availability depends on GPU/driver/toolchain, making it difficult to include as a stable default in the reproducible mainline.

Fused CUDA residual add. We implemented and rigorously validated a fused CUDA residual-add kernel for GPT-NeoX (bit-exact hidden states and generation outputs), but it was slower (0.92×, TPOT +8.6%), consistent with Amdahl's law and kernel-launch dominated micro-ops.

MIT reference implementation notes. The MIT StreamingLLM reference repo applies `pos_shift` patches inside model attention forwards, while our implementation preserves HuggingFace attention modules and performs RoPE consistency as cache-side operations. Under our Transformers version, we observed that the MIT long-PPL script did not produce consistent metrics for GPT-NeoX; therefore we avoid drawing quality claims from cross-codebase PPL and focus on within-evaluator comparisons.

References

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.

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

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.

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

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

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.

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

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.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

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.

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

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.