FLASHLOOP: FAST AND MEMORY-EFFICIENT LOOPED TRANSFORMERS VIA LAZY UPDATES

Anonymous authors
Paper under double-blind review

ABSTRACT

Looped Transformers have attracted substantial attention as a parameter-efficient approach to increasing computational depth through repeated application of shared Transformer blocks. However, their practical advantages over conventional Transformers remain under debate: each additional loop incurs another Transformer pass and requires caching another set of KV states, causing inference FLOPs and KV-cache memory to grow continuously with loop depth. This overhead becomes particularly severe at large loop counts and long context, preventing the parameter efficiency of Looped Transformers from translating into practical inference efficiency. In this paper, we find that much of the additional computation and storage introduced by looping is redundant. As recurrence proceeds, state changes become increasingly concentrated on a small subset of tokens; attention-output differences are dominated by a sparse and stable subset of key columns; and KV residuals between adjacent loops become progressively more amenable to low-bit quantization. Building on these observations, we introduce **FlashLoop**, a training-free inference framework that reduces cross-loop redundancy through token-sparse updates, sparse attention, and KV-residual quantization. Across several Looped Transformers models, FLASHLOOP delivers lossless accuracy while achieving up to **1.64**$\times$ end-to-end speedup and up to **6**$\times$ KV-cache memory reduction, substantially improving the practicality of scaling Looped Transformers to greater computational depths and longer context.

1 INTRODUCTION

Looped Transformers offer a promising way to decouple a model's computational depth from its parameter count. By repeatedly applying a shared Transformer module, they allow relatively compact models to perform substantially deeper computation while keeping the stored model weights fixed (Dehghani et al., 2018; Zhu et al., 2025; Geiping et al., 2026). This opens a new scaling dimension: increasing test-time compute and thereby computational depth, without increasing the model's parameter-memory footprint.

However, a lower parameter count does not necessarily translate into lower inference costs. Unrolling an N-layer looped Transformer with R loops requires $R \times N$ block executions and a distinct KV cache per iteration, making computation and KV-cache memory grow linearly with loop depth. On a single A100 machine, the prefill time for a 32K-token prompt is 27 seconds for Ouro-2.6B (Zhu et al., 2025) with four loops, compared with only 3 seconds for LLaMA-3.1-8B (Grattafiori et al., 2024). Meanwhile, Ouro's KV cache alone requires 48 GiB, compared with just 4 GiB for LLaMA-3.1-8B. Thus, despite having roughly one-third as many parameters, the looped model can impose substantially greater latency and memory requirements.

To uncover opportunities for reducing this growing inference overhead, we systematically characterize the cross-loop dynamics of representative looped Transformers. We observe a pronounced *lazy-update pattern* in their inference dynamics: as recurrence progresses, changes between adjacent loops become increasingly sparse, predictable, and compressible. This pattern gives rise to three major forms of redundancy:

- Token-update redundancy. Updates to most token states become increasingly small, with only a small fraction of token rows accounting for the majority of changes in hidden states and KV representations.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

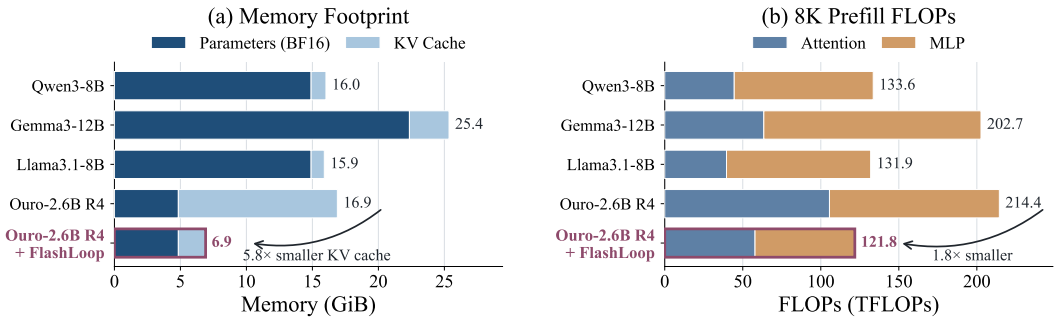


Figure 1: **Comparison of performance-matched looped and non-looped models at an 8K context length**[1]**.** FlashLoop substantially reduces both computation and KV-cache memory for looped Transformers, achieving 5.8× and 1.8× reductions in KV cache memory and 8K prefill FLOPs.

- Attention-computation redundancy. Cross-loop differences in attention outputs are dominated by a sparse subset of columns whose indices can be reliably predicted from the preceding loop.
- KV-storage redundancy. KV residuals between adjacent loops are substantially more friendly to quantization than full KV states, exhibiting lower reconstruction error under identical quantization settings.

Building on these observations, we introduce **FlashLoop**, a training-free and loop-native inference framework that reduces cross-loop redundancy through three complementary components. First, *Token-Sparse Updates* reuses the hidden and KV states of converged tokens, recomputing states and storing new KV entries only for active tokens. This reduces computation and avoids maintaining a complete KV cache for every loop. Second, *Sparse Attention* selects important keys using attention-contribution statistics from the preceding loop, while cached global probability mass preserves the full-softmax scale. Third, *KV-Residual Quantization* compresses the cache into a quantized base state and low-bit residual updates between successive loops. We implement these components efficiently on GPUs through hardware-aware pipelining and optimized kernels.

Across several Looped Transformers models, FlashLoop retains the parameter efficiency of looped Transformers and largely preserves their accuracy, while delivering up to $1.64\times$ end-to-end speedup and reducing KV-cache memory by up to $6\times$. As illustrated in Figure 1, these improvements bring the inference costs of looped transformers to levels comparable to, or lower than, those of performance-matched non-looped models.

2 Related Work

Looped Transformers and loop-specific KV-cache reduction. The Universal Transformer introduced depth recurrence and adaptive per-position computation to combine the parallelism of Transformers with the inductive biases of recurrent models (Dehghani et al., 2018). Subsequent recurrent language models develop this idea in different ways. Chain-of-Experts sequentially reuses FFN experts with iteration-specific routing to increase effective computational depth (Wang et al., 2025). Huginn repeatedly applies a shared recurrent module and treats the recurrence count as a potential control over test-time compute (Geiping et al., 2026). Ouro instead incorporates latent reasoning capacity into pretraining through weight-shared recurrence, showing high parameter efficiency (Zhu et al., 2025). Several methods have been proposed to manage the resulting loop-indexed KV cache. Memory-Efficient Looped Transformer reduces cache overhead through learned gated updates, whereas Looped Latent Attention trains encoders and decoders to construct compressed representations along the recurrence dimension (Vendrell et al., 2026; Neill & Reid, 2026). However, these methods either modify the model architecture or require additional training. In contrast, FlashLoop starts from a pretrained recurrent model, eliminates cross-loop structural redundancy at inference time without modifying model weights or requiring any training.

[1]Accuracy results for these models are reported in Appendix A.

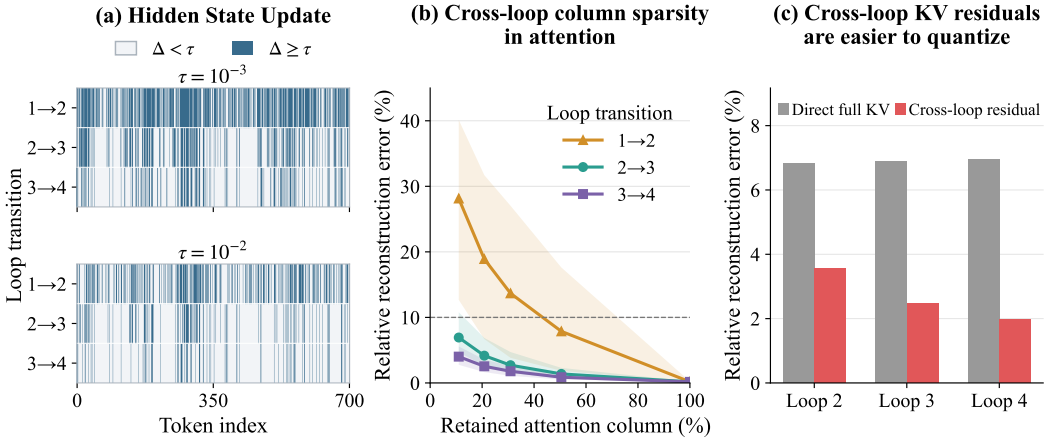


Figure 2: **Recurrent refinement becomes structurally redundant along three complementary axes.** (a) *Token-level cross-loop redundancy*: hidden-state change becomes increasingly concentrated at a subset of tokens. (b) *Attention-column redundancy*: later-loop attention outputs can be reconstructed from a small subset of key columns identified by the preceding loop. (c) *Cross-loop representation redundancy*: at the same quantization setting, recursively quantizing adjacent-loop KV residuals gives lower reconstruction error than independently quantizing complete KV states.

Generic inference optimization. Existing general inference optimization methods reduce computational and memory overhead along several complementary dimensions. Quantization lowers the storage and bandwidth costs of the parameters and KV cache (Liu et al., 2024b; Hooper et al., 2024; Yang et al., 2026). KV cache sharing merges or reuses keys and values by exploiting similarities or redundancies across network layers (Liu et al., 2024a; Brandon et al., 2024). Token and KV eviction controls the effective context length and cache overhead by retaining only the historical tokens deemed most important for subsequent generation (Liu et al., 2023; Zhang et al., 2023; Li et al., 2024; Cai et al., 2024). Sparse attention reduces attention computation and KV reads through structured attention patterns or query-dependent selection (Ribar et al., 2023; Jiang et al., 2024; Tang et al., 2024). Collectively, these methods primarily target non-looped Transformers and therefore do not explicitly exploit structural redundancy along the recurrence axis of looped Transformers. In contrast, FlashLoop identifies loop-specific, multilevel redundancy arising during looped execution and provides targeted optimizations for this redundancy, making it complementary to existing general inference optimization approaches.

3 Cross-Loop Redundancy in Looped Transformers

Let one recurrent computation of the looped Transformer parameters be defined as

$$H^{(r)} = F_\theta\left(H^{(r-1)}\right), \tag{1}$$

where $r \in 1, \ldots, R$ denotes the loop index, $H^{(r)}$ denotes the token hidden states after the r-th loop, and F_θ denotes the Transformer block parameterized by θ and shared across different loops. In the original implementation, F_θ is applied to all tokens at every loop, while a separate pair of keys and values, $\left(K^{(r)}, V^{(r)}\right)$, is stored for each loop, which means the computation and KV cache increase linearly with loop count.

This execution pattern, however, treats every loop as a full update, overlooking the increasingly structured changes between successive loops. Specifically, substantial hidden-state changes become concentrated in a small subset of token rows, important attention columns remain concentrated and stable across loops, and adjacent-loop KV residuals are more amenable to quantization than full KV states. Figure 2 and Appendix B summarize these three forms of cross-loop structural redundancy.

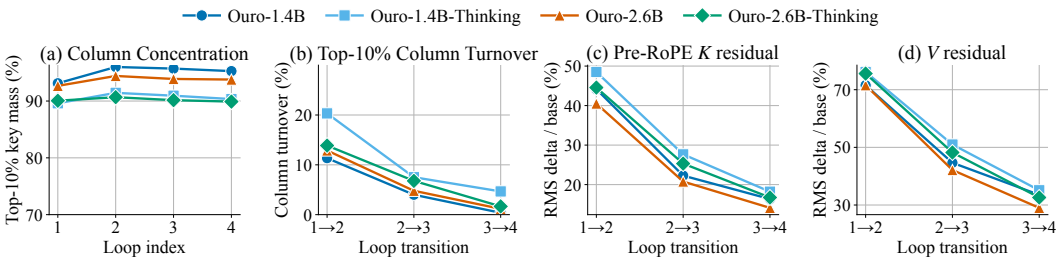


Figure 3: **Additional evidence for cross-loop redundancy across different looped models.** (a) Different looped transformers have similar cross-loop redundancy: decode attention concentrates on a small set of attention columns, (b) whose importance becomes increasingly stable across loops, (c–d) while adjacent-loop K/V residuals decrease with loop depth motivating a residual-based quantization for cross-loop KV states.

3.1 Token-Update Redundancy

The first form of redundancy we observe is that iterative refinement is non-uniform across token positions in the context: the hidden states of different tokens converge at different rates across loops. Figure 2(a) illustrates two key characteristics of token-level convergence across loops. First, tokens with substantial hidden-state changes become increasingly sparse as recurrence proceeds, while remaining distributed throughout the context. Second, these high-change tokens exhibit an approximately nested structure: tokens that remain active in a later loop transition are largely a subset of those active in the preceding transition.

Since keys and values are computed independently from each token representation, this token-level sparsity of hidden-state updates naturally induces sparsity in KV updates. Thus, tokens whose hidden states have effectively converged can directly reuse their KV states from the preceding loop. This progressive concentration of token-level state changes motivates the sparse computation mechanism in FlashLoop, which selectively shares KV cache across loops and updates tokens based on their cross-loop changes.

3.2 Attention-Computation Redundancy

The second form of redundancy we observe lies in the cross-loop updates of the attention output. Let's revisit the equation of attention:

$$o^{(r)} = \mathrm{softmax}\left(\frac{q^{(r)} {K^{(r)}}^\top}{\sqrt{d}}\right) V^{(r)}. \tag{2}$$

Each key/value index corresponds to a column of the attention matrix across queries, whose attention weights scale the associated value vector in the output aggregation. We refer to this key/value-specific contribution as an *attention column*. We find that cross-loop changes in the attention output are concentrated on a sparse subset of attention columns: most change only slightly, while a small subset accounts for most of the change in the attention output, as shown in Figure 3(a). Moreover, the importance of these columns remains stable across adjacent loops (Figure 3(b), Figure 9), allowing the previous loop to identify the columns that are likely to remain important in the next loop.

This observation suggests approximating the current-loop attention output with sparse column-wise updates to the previous-loop output. Leveraging the cross-loop stability of column importance, we rank key columns by their attention probabilities in loop $r-1$ and let S denote the indices of the top-K columns. We then reconstruct the attention output of loop r by replacing only these columns' contributions in $o^{(r-1)}$:

$$\widehat{o}^{(r)} = o^{(r-1)} - m\left[\mathrm{softmax}\left(\frac{q^{(r-1)} {K_S^{(r-1)}}^\top}{\sqrt{d}}\right)\right] V_S^{(r-1)} + m\left[\mathrm{softmax}\left(\frac{q^{(r)} {K_S^{(r)}}^\top}{\sqrt{d}}\right)\right] V_S^{(r)} \tag{3}$$

where m is a renormalization factor used to preserve the global attention mass of the selected columns. The selected columns are recomputed using the current-loop states, while the remaining contributions

are reused from the previous loop. Figure 2(b) shows that the reconstruction error decreases as more columns are retained and, for the same retention rate, becomes substantially lower at later loop transitions. In loop 3/4 of Ouro-1.4B, updating only 10% of the columns can reconstruct over 90% of exact attention output.

3.3 Quantization-Friendly KV Residual

Although our previous observations allow a subset of tokens to be set as inactive in the middle and later loops and reuse their previously generated KV cache, active tokens continue to produce new KV cache in both early and later loops. In long-context settings, caching these KV states accounts for a major portion of the memory overhead.

To compress this part of the KV cache through quantization, we analyze its quantization friendliness of the full KV states and the KV residual between the current loop and their preceding loop. As shown in Figure 3(c,d), the magnitudes of these residuals decrease with loop depth, making the incremental updates increasingly small relative to the full KV states and thus more sensitive to quantization error when the full states are quantized directly. The results in Figure 2(c) show that quantizing these cross-loop residuals yields substantially lower relative reconstruction error than directly quantizing the full KV states, and this advantage becomes more pronounced as the loop depth increases. These observations motivate representing the loop-wise KV cache as a quantized base state with quantized cross-loop residual updates.

4 FlashLoop

To reduce these structural redundancies, we propose FlashLoop, a training-free and loop-native inference framework for pretrained looped Transformers. FlashLoop includes three coupled compression strategies: token-sparse updates, sparse attention, and KV residual quantization. We also provide the mathematical formulation of our framework in Appendix D.

4.1 Cross-Loop Token-Sparse Updates

Motivated by the progressive concentration of hidden-state changes observed in Section 3.1, FlashLoop selectively updates only the tokens that remain active across loops. After the dense early loops, each subsequent loop ranks the tokens updated in the preceding loop according to their normalized hidden-state changes and retains only the top-k tokens for further refinement, where k can vary with loop depth. Since skipped tokens do not produce new hidden-state changes, later selections are restricted to the active tokens from the preceding loop, naturally yielding nested active-token sets.

This design reduces both computation and memory overhead. For each sparse loop, the looped Transformer block performs computation only on the selected active tokens, which avoids wasting computation on those inactive tokens. The same token-level decision also applies to KV-cache updates. Active tokens generate new K/V states for the current loop, whereas inactive tokens continue to reuse their KV cache from preceding loop. Consequently, a later-loop KV cache does not require another complete copy of all token states and saves memory for that.

4.2 Loop-Aware Sparse Attention

Building on the cross-loop sparsity and stability of important attention columns identified in Section 3.2, FlashLoop uses the preceding loop to identify a small subset of columns for recomputation. For each sparse loop, we rank the attention columns using statistics from the previous loop and select the top-K columns under a loop-specific budget. We then gather only their corresponding key/value entries, allowing both the QK product and value aggregation to operate on this reduced set.

A direct softmax over the selected columns, however, would renormalize their probabilities to sum to one and therefore overestimate their contribution to the full attention output. Thus, FlashLoop caches their *global probability mass*, i.e., the total probability assigned to the selected columns under the full attention distribution of the preceding loop. In the target loop, we apply softmax only over the selected logits and rescale the resulting probabilities by this cached mass, approximately preserving their contribution under the full attention distribution.

The rescaled value contributions are then used to update the attention output according to Eq. 3. When later loops use smaller column budgets, we further prune the selected set using attention statistics from the preceding sparse loop. Both the selected column indices and their associated probability mass are propagated across loops, enabling progressively sparser attention computation.

4.3 Cross-Loop KV Residual Quantization

As observed in Section 3.3, cross-loop KV residuals are more quantization-friendly than full KV states. FlashLoop therefore represents the KV cache as a quantized base state followed by quantized residual updates across loops. The K/V states from the first loop are quantized as the base, while each subsequent loop stores only the residual relative to the reconstructed state of the preceding loop. For $X \in \{K, V\}$, we define

$$\begin{aligned} \widehat{X}^{(1)} &= D\Big(Q\Big(X^{(1)}\Big)\Big), \qquad \Delta X^{(r)} = X^{(r)} - \widehat{X}^{(r-1)}, \\ \widehat{X}^{(r)} &= \widehat{X}^{(r-1)} + D\Big(Q\Big(\Delta X^{(r)}\Big)\Big), \qquad 2 \le r \le R, \end{aligned} \tag{4}$$

where $Q(\cdot)$ and $D(\cdot)$ denote quantization and dequantization, respectively. At read time, the required K/V state is reconstructed by accumulating the quantized residual updates on top of the base state.

Cross-loop KV residual quantization naturally complements our cross-loop token-sparse updates. In sparse loops, inactive tokens reuse their existing K/V states, while residuals are generated and stored only for active tokens. FlashLoop therefore reduces both the number of new K/V entries stored across loops and the storage and bandwidth cost of each remaining update.

4.4 Fast GPU Support

Although the blueprint described above is simple at the algorithmic level, efficiently realizing sparsity and low-bit KV residual on GPUs requires careful system support. Dynamic token and column selection can introduce irregular memory accesses and fragmented computation, while separate cache-update and quantization steps may add additional data movement and kernel-launch overhead. FlashLoop addresses these issues by gathering selected tokens and keys into compact, compute-friendly buffers to preserve regular GEMM execution, and by fusing cross-loop KV updates and quantization into the attention kernel. These optimizations allow FlashLoop to translate its algorithmic sparsity and compression into practical reductions in computation and KV-cache memory. We provide more details in Appendix G.

5 Experiments

5.1 Experimental Setup

We evaluate FlashLoop on five representative Looped Transformers: four models from the Ouro (Zhu et al., 2025) family and Huginn-3.5B (Geiping et al., 2026). We evaluate FlashLoop on five benchmarks: MATH-500 (Lightman et al., 2023), GSM8K (Cobbe et al., 2021), ARC-Challenge (Clark et al., 2018), HellaSwag (Zellers et al., 2019) and WinoGrande (Sakaguchi et al., 2019), using the EleutherAI LM Harness (Gao et al., 2024). Moreover, we provide the detailed setting such as token and column sparsity settings for each loop of each model and the quantization setting in Appendix E.

5.2 Main Results

We report our results in Table 1. Overall, FlashLoop largely preserves the task performance of the original models while substantially reducing their inference costs. Moreover, across model variants, base models tend to be more robust to FlashLoop than their thinking counterparts, suggesting that thinking models may be more sensitive to cross-loop compression. Nevertheless, FlashLoop still largely preserves the performance of the thinking models, indicating that substantial cross-loop redundancy can be exploited even in these models.

Table 1: **Task performance and efficiency of Ouro and Huginn with FlashLoop.** We report accuracy (%) on five benchmarks. Δ Avg. denotes the average-score difference from the original model in percentage points. KV Memory means the reduction of peak KV cache memory relative to original models.

Model	Method	MATH-500	ARC-C	GSM8K	HellaSwag	WinoGrande	Avg.	Δ Avg.	Speedup ↑	KV Memory ↑
Ouro-1.4B	Original	65.80	59.98	78.77	74.24	71.74	70.11	–	1.00×	1.00×
	FlashLoop	65.40	60.67	79.61	74.33	72.38	70.48	+0.37	1.59×	5.85×
Ouro-1.4B-Thinking	Original	46.80	62.03	81.65	72.61	72.45	67.11	–	1.00×	1.00×
	FlashLoop	46.20	61.69	80.39	72.38	70.51	66.23	−0.88	1.59×	5.85×
Ouro-2.6B	Original	52.20	66.13	81.80	79.54	76.40	71.21	–	1.00×	1.00×
	FlashLoop	52.80	65.10	82.56	79.58	75.37	71.08	−0.13	1.64×	6.06×
Ouro-2.6B-Thinking	Original	59.00	66.21	83.47	78.22	76.09	72.60	–	1.00×	1.00×
	FlashLoop	55.60	67.38	83.94	79.25	75.03	72.24	−0.36	1.64×	6.06×
Huginn-3.5B	Original	11.00	43.94	27.52	68.01	62.51	42.60	–	1.00×	1.00×
	FlashLoop	13.40	44.45	25.70	67.95	62.35	42.77	+0.17	1.52×	5.18×

Table 2: **Component ablation of FlashLoop on Ouro-1.4B.** FlashLoop w/ Per-loop KIVI4 means FlashLoop with full KV cache quantization along the same quantization axis. All values are accuracy (%).

Method	MATH-500	ARC-C	GSM8K	Avg.
Original	65.80	59.98	78.77	68.18
Loop-aware Sparse Attention	69.00	60.24	77.48	68.91
+ Cross-Loop Token-Sparse Updates	66.60	60.07	79.23	68.63
FlashLoop w/ Per-loop KIVI4	64.40	59.87	77.10	67.12
FlashLoop	65.40	60.67	79.61	68.56

Table 3: **Comparison with alternative KV-cache strategies.** We report average accuracy (%) across five benchmarks. "T" represents thinking version.

Model	Original	H_2O (-75% KV)	Last-step (-75% KV)	FlashLoop
Ouro-1.4B	70.11	62.81	68.33	**70.48**
Ouro-1.4B-T	67.11	62.68	65.09	**66.23**
Ouro-2.6B	71.21	67.35	69.96	**71.08**
Ouro-2.6B-T	72.60	68.01	70.78	**72.24**
Huginn-3.5B	42.60	40.68	32.18	**42.77**

5.3 Ablation Study

5.3.1 Contribution of Each Component

We conduct a series of ablation experiments to assess the contribution of each component in FlashLoop. Our goal is to maximize efficiency while preserving the accuracy of looped Transformers, so we evaluate each component's impact on accuracy and its contribution to efficiency.

Accuracy. Table 2 reports the accuracy ablation results. Loop-aware sparse attention alone slightly improves the average performance over the original model. Adding token-sparse updates largely preserves this gain and remains competitive across all three benchmarks, suggesting that the introduced sparsity does not compromise model accuracy. KV residual quantization introduces a small accuracy drop, while the full FlashLoop configuration still maintains performance comparable to the original model. We further ablate the design of KV quantization by replacing cross-loop residual quantization with direct quantization of the full KV states under the same quantization setting. This alternative leads to a noticeably larger accuracy degradation, consistent with our observation in Section 3.3 that cross-loop KV residuals are more quantization-friendly than full KV states.

Efficiency. Figure 4 reports the efficiency ablation results and shows the distinct benefits of the three components. Cross-loop token-sparse updates substantially reduce prefill latency and KV-cache storage by recomputing only token states that continue to change. Loop-aware sparse attention then offsets much of the resulting decode-latency increase by restricting computation to important key columns. Finally, 4-bit KV residual quantization further reduces cache footprint and decode latency by lowering KV traffic. Although quantization adds work during prefill, the complete FlashLoop configuration is faster than the original model in both prefill and decoding while using substantially less KV memory.

5.3.2 Comparison with Alternative KV-Cache Strategies

We further compare FlashLoop with two alternative KV-cache compression strategies. H_2O (Zhang et al., 2023) retains 25% of cached tokens based on their accumulated attention scores, while Last-step

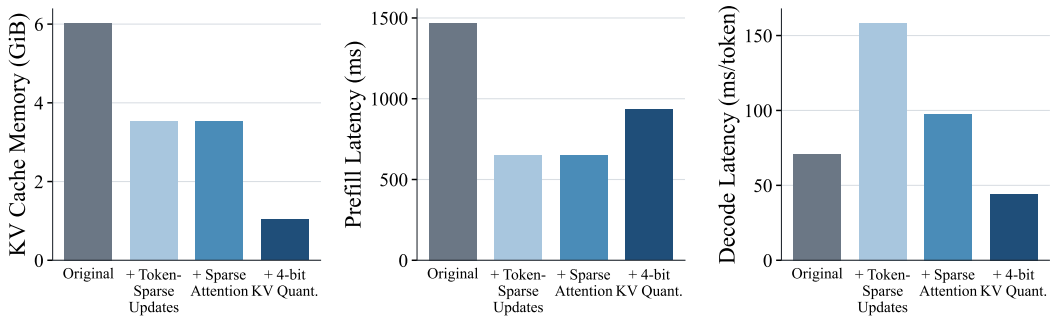


Figure 4: **Efficiency analysis of each component.** We cumulatively enable cross-loop token-sparse updates, loop-aware sparse attention, and 4-bit KV residual quantization, reporting KV-cache memory, prefill latency, and decode latency.

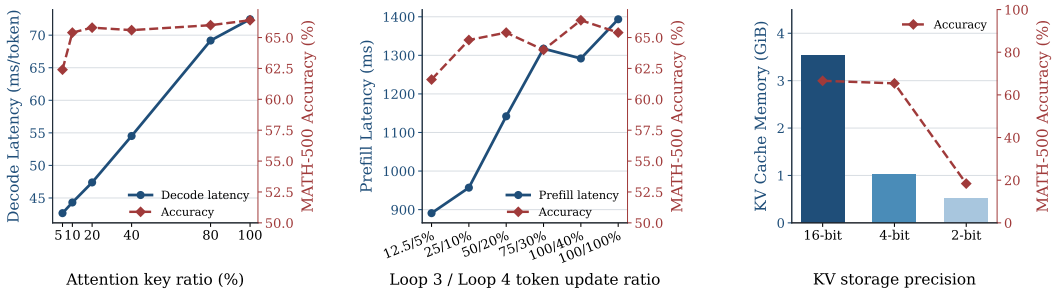


Figure 5: **Accuracy–efficiency trade-offs of FlashLoop hyperparameters on Ouro-1.4B.** We vary the retained attention-key ratio (left), Loop-3/4 token-update ratios (middle), and KV residual quantization precision (right), while keeping the remaining components fixed.

KV Reuse (Zhu et al., 2025) retains only the final-loop KV cache for subsequent decoding, while for Huginn-3.5B we reuse last-step KV of every four steps.

Table 3 reports the average accuracy across the five benchmarks in Table 1. H_2O compresses each loop independently and therefore does not exploit the structural redundancy across loops, leading to substantial performance degradation. Last-step KV Reuse leverages the recurrent structure but discards the distinct information preserved in intermediate-loop KV states, resulting in additional accuracy loss. FlashLoop instead preserves useful cross-loop information while selectively compressing redundant updates, allowing it to maintain performance close to the original models under substantial KV-cache compression.

5.3.3 Sparsity and Quantization Configurations

We conduct parameter ablations to evaluate the sensitivity of FlashLoop to the sparsity and quantization settings of its three components and summarize the resulting accuracy–efficiency trade-offs.

As reported in Figure 5, model accuracy is robust to attention-column sparsity across most evaluated configurations, with noticeable degradation only at very aggressive column sparsity. It remains nearly unchanged from the accuracy of the original model except at 5% column retention. The model is similarly robust to our cross-loop token-sparse updates: retaining 25% of tokens in loop 3 and 10% in loop 4 can maintain accuracy comparable to that of the original model, while keeping system overhead very low. Moreover, compared with BF16, 4-bit KV residual quantization substantially reduces the KV cache footprint and decode kernel latency, with negligible impact on accuracy, whereas 2-bit setting suffers from clear performance degradation.

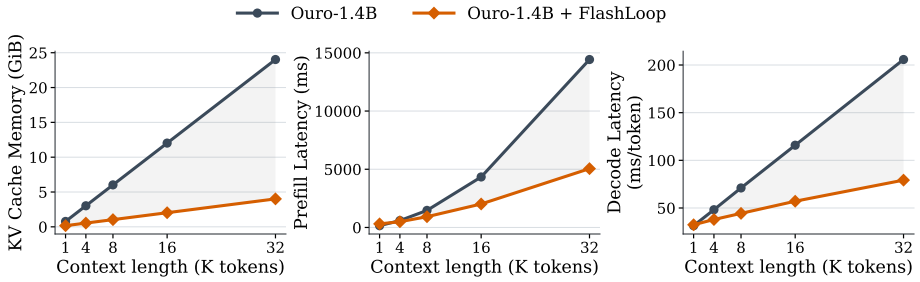


Figure 6: **Scaling with context length.** FlashLoop delivers increasingly substantial memory savings and speedups as the context length grows.

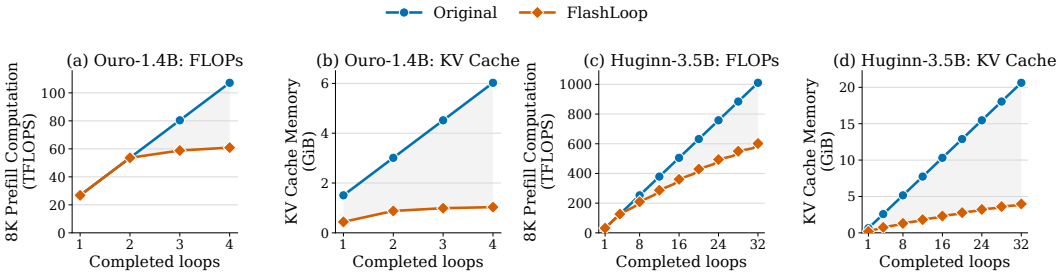


Figure 7: **Scaling with loops.** As the number of loops increases, FlashLoop progressively reduces the accumulated prefill FLOPs and KV-cache memory of both Ouro-1.4B and Huginn-3.5B.

5.4 End-to-end Efficiency Analysis

We compare baseline inference with the full FlashLoop across context lengths and loops on a single NVIDIA A100-SXM4-40GB GPU. To improve measurement stability, we repeat prefill three times after warm-up and report the median time measured using CUDA events. For decode, we use 16 tokens for warm-up and measure the average per-token latency over the next 16 tokens.

Scaling with Context Length. As shown in Figure 6, the benefits of FlashLoop become more pronounced as the context length increases. At 32K context length, FlashLoop reduces KV-cache storage by 20 GiB, corresponding to an 82.8% reduction, while achieving a significant speedup in prefill and decode. We also provide a long-context quality evaluation in Appendix F.

Scaling with Loops. Figure 7 shows how computation and KV-cache memory scale with the number of loops for Ouro-1.4B and Huginn-3.5B at an 8K context length. For the original models, both quantities grow approximately linearly with loop count. In contrast, FlashLoop substantially slows this growth in later loops, where only a subset of tokens and attention columns are updated and less additional KV cache is stored. As a result, the efficiency gains become increasingly pronounced as more loops are executed, particularly for Huginn-3.5B with 32 loops. These results demonstrate that FlashLoop effectively mitigates the inference overhead associated with deep recurrence.

6 Conclusion

In this work, we identify three forms of cross-loop redundancy in looped Transformers, and based on these observations, we propose FlashLoop, a training-free inference framework that exploits such redundancy through cross-loop token-sparse updates, loop-aware sparse attention, and cross-loop KV residual quantization. Across multiple looped models and benchmarks, FlashLoop substantially reduces computation and KV-cache storage while largely preserving the task performance of the original models, while these gains become more pronounced with longer contexts and deeper recurrence.

AI USE STATEMENT

Generative AI tools were used to improve the readability of the manuscript and to assist with minor code editing. All AI-assisted text and code were reviewed and verified by the authors. Generative AI tools were not used to formulate the research hypotheses, design the methodology, interpret the experimental results, or derive the main scientific claims. The authors take full responsibility for the content of this work.

REPRODUCIBILITY STATEMENT

We provide implementation details, compression configurations, and evaluation protocols in the main text and appendix. Our code is included in the supplementary material. All experiments use publicly available models and benchmarks.

ETHICS STATEMENT

This work studies inference efficiency in Looped Transformers using publicly available models and benchmarks. FlashLoop does not address biases or harmful-generation risks inherited from the underlying models. More efficient inference may facilitate both beneficial applications and misuse, deployments should therefore retain appropriate safety safeguards.

REFERENCES

William Brandon, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan Kelly. Reducing transformer key-value cache size with cross-layer attention, 2024. *URL https://arxiv. org/abs/2405.12981*, 2024.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, et al. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling, 2024. *URL https://arxiv. org/abs/2406.02069*, 2024.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv:1803.05457v1*, 2018.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024. URL `https://zenodo.org/records/12608602`.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. *Advances in Neural Information Processing Systems*, 38: 41340–41391, 2026.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems*, 37:1270–1303, 2024.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H Abdi, Dongsheng Li, Chin-Yew Lin, et al. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *Advances in Neural Information Processing Systems*, 37:52481–52515, 2024.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 37:22947–22970, 2024.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Gholamreza Haffari, and Bohan Zhuang. Minicache: Kv cache compression in depth dimension for large language models. *Advances in Neural Information Processing Systems*, 37:139997–140031, 2024a.

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. *Advances in Neural Information Processing Systems*, 36:52342–52364, 2023.

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

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

James O' Neill and Fergal Reid. Looped latent attention: Cross-loop kv compression for looped transformers. *arXiv preprint arXiv:2607.15456*, 2026.

Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. Sparq attention: Bandwidth-efficient llm inference. *arXiv preprint arXiv:2312.04985*, 2023.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *arXiv preprint arXiv:1907.10641*, 2019.

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

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

Victor Conchello Vendrell, Arnau Padres Masdemont, Niccolò Grillo, Jordi Ros-Giralt, Arash Behboodi, and Fabio Valerio Massoli. Memory-efficient looped transformer: Decoupling compute from memory in looped language models. *arXiv preprint arXiv:2605.07721*, 2026.

Zihan Wang, Rui Pan, Jiarui Yao, Robert Csordas, Linjie Li, Lu Yin, Jiajun Wu, Tong Zhang, Manling Li, and Shiwei Liu. Chain-of-experts: unlocking the communication power of mixture-of-experts models. *arXiv preprint arXiv:2506.18945*, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Wanqi Yang, Yuexiao Ma, Alexander Conzelmann, Xiawu Zheng, Michael W Mahoney, T Konstantin Rusch, and Shiwei Liu. Alphaq: Calibration-free bit allocation for mixture-of-experts quantization. *arXiv preprint arXiv:2606.04980*, 2026.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

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. *Advances in neural information processing systems*, 36:34661–34710, 2023.

Rui-Jie Zhu, Zixuan Wang, Kai Hua, Tianyu Zhang, Ziniu Li, Haoran Que, Boyi Wei, Zixin Wen, Fan Yin, He Xing, et al. Scaling latent reasoning via looped language models. *arXiv preprint arXiv:2510.25741*, 2025.

Table 4: **Benchmark results of open-sourced performance-matched looped and non-looped models.** All models are base models. The best score in each column is **bolded**, and the second-best is underlined.

Model	ARC-C	HellaSwag	WinoGrande	GSM8K	MATH-500
Ouro-2.6B R4	66.13	79.54	76.40	81.80	52.20
Qwen3-8B	66.10	79.60	76.80	**83.09**	62.30
Gemma3-12B	**72.44**	**83.68**	**77.74**	77.18	**63.20**
Llama-3.1-8B	60.75	81.97	77.11	78.17	52.90

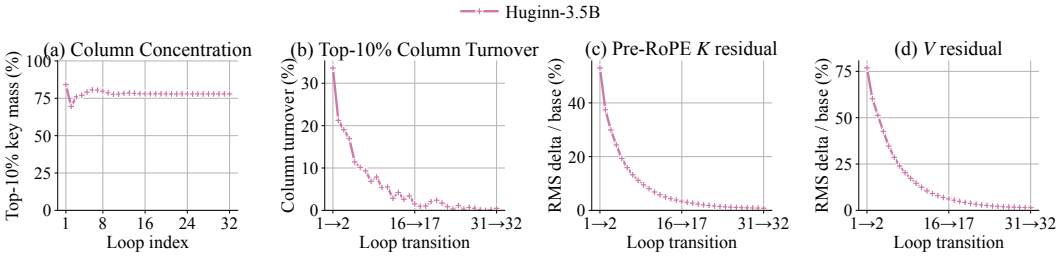


Figure 8: **Cross-loop redundancy in Huginn-3.5B.** We record the same metrics of Huginn-3.5B as in Figure 3, which show similar redundancy as in Ouro family.

A Performance-matched Looped and Non-looped Models

We report the accuracy of performance-matched looped and non-looped large language models, including Ouro-2.6B (Zhu et al., 2025) with four loops, Qwen3-8B (Yang et al., 2025), Gemma3-12B (Team et al., 2025) and Llama-3.1-8B (Grattafiori et al., 2024). The results in Table 4 and Figure 1 illustrate that Ouro-2.6B with four loops achieves comparable accuracy in several benchmarks, but its inference efficiency is still a challenge because of its KV cache memory overhead and computation. FlashLoop effectively resolves this challenge without training.

B Cross-loop redundancy in model with more loops

We also study Huginn-3.5B with 32 recurrent loops and observe similar cross-loop redundancy and attention concentration. As shown in Figure 8, a small fraction of columns captures most of the attention probability mass. In later loops, these important columns become highly stable, while the KV residuals between adjacent loops shrink substantially. These observations show that the structure exploited by FlashLoop also appears in models with much deeper recurrent computation.

C Attention Patterns Across Loops

Figure 9 visualizes prefill attention probabilities across four loops of Ouro-1.4B. Within the same layer, important key-column patterns recur across loops, and the columns stay stable across loops. This cross-loop persistence motivates FlashLoop to use the preceding loop's attention probabilities to select the columns updated in the next loop.

D Mathematical Formulation of FlashLoop

We provide the mathematical formulation of our method. In our formulation, H, K, and V denote hidden states, keys, and values. We use r to index loops and i, j to index token positions.

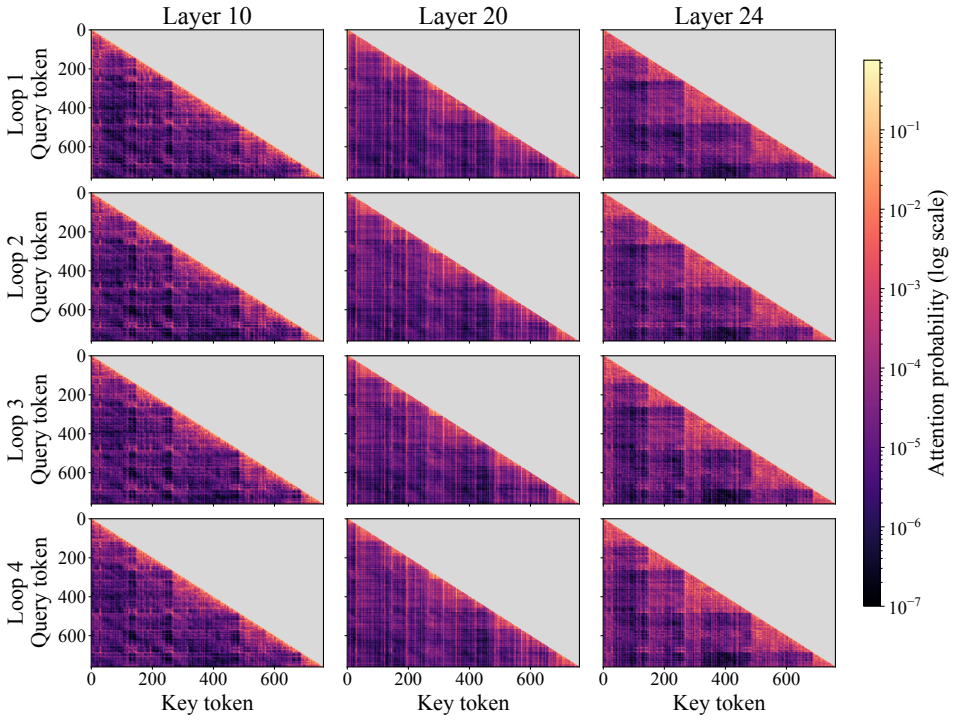


Figure 9: **Prefill attention maps for a random sampled MATH-500 prompt.** There are obvious important key-column patterns across different layers and loops, while the important columns are stable across loops.

D.1 Cross-Loop Token-Sparse Updates

After the dense early loops, FlashLoop selects active tokens using their normalized hidden-state changes in the preceding loop:

$$s_i = \frac{\|H_i^{(r-1)} - H_i^{(r-2)}\|_2}{\|H_i^{(r-1)}\|_2 + \epsilon}, \qquad A^{(r)} = \mathrm{TopK}_{k^{(r)}}\big(\{s_i : i \in A^{(r-1)}\}\big), \tag{5}$$

where $A^{(r)}$ is the active-token set, $k^{(r)}$ is its budget, and $\epsilon > 0$ is a small constant. After dense early loops update all tokens, later loops recompute only $A^{(r)} \subseteq A^{(r-1)}$, while the remaining tokens reuse their hidden states and KV cache.

D.2 Loop-aware Sparse Attention

For the current decode position, d denotes the dimension of each attention head, $\mathcal{C}$ denotes the keys computed in loop $r-1$, and $m_{\mathcal{C}}$ denotes their cached global mass. During loop $r-1$, FlashLoop computes attention probabilities $\widehat{p}^{(r-1)}$ and uses the known next-loop budget $c^{(r)}$ to select and cache the selected index subset S and its global mass m:

$$\widehat{p}_j^{(r-1)} = m_{\mathcal{C}} \frac{\exp\Big(q^{(r-1)}\big(\widehat{K}_j^{(r-1)}\big)^\top/\sqrt{d}\Big)}{\sum_{i\in\mathcal{C}} \exp\Big(q^{(r-1)}\big(\widehat{K}_i^{(r-1)}\big)^\top/\sqrt{d}\Big)}, \quad j \in \mathcal{C}, \qquad S = \mathrm{TopK}_{c^{(r)}}\big(\{\widehat{p}_j^{(r-1)} : j \in \mathcal{C}\}\big), \qquad m = \sum_{j\in S} \widehat{p}_j^{(r-1)}, \tag{6}$$

where a dense source loop uses all causal keys and $m_{\mathcal{C}} = 1$. A sparse source loop uses its selected keys and the mass cached for that selection, where the budget satisfies $1 \le c^{(r)} \le |\mathcal{C}|$. In loop r,

FlashLoop reuses the cached subset S and mass m to compute

$$\widehat{p}_S^{(r)} = m \,\mathrm{softmax}\left(\frac{q^{(r)}\big(\widehat{K}_S^{(r)}\big)^\top}{\sqrt{d}}\right), \qquad \widehat{o}^{(r)} = \widehat{o}^{(r-1)} - \widehat{p}_S^{(r-1)}\widehat{V}_S^{(r-1)} + \widehat{p}_S^{(r)}\widehat{V}_S^{(r)}, \tag{7}$$

where $q^{(r)}$ is the current query. Within loop r, the updated probabilities likewise prepare the nested selection and cached mass for loop $r+1$. The reconstruction results in Figure 2(b) use this mass-corrected formulation.

D.3 Cross-Loop KV Residual Quantization

As described in Section 4.3, for $X \in \{K, V\}$, FlashLoop stores a quantized base and cross-loop residuals. Let Q and D denote quantization and dequantization. The first loop provides $\widehat{X}^{(1)} = D(Q(X^{(1)}))$. For later loops, we calculate $\widehat{X}_i^{(r)}$, the reconstructed KV used by sparse attention, as follows:

$$\Delta X_i^{(r)} = X_i^{(r)} - \widehat{X}_i^{(r-1)}, \qquad i \in A^{(r)}, \qquad \widehat{X}_i^{(r)} = \begin{cases} \widehat{X}_i^{(r-1)} + D\big(Q\big(\Delta X^{(r)}\big)\big)_i, & i \in A^{(r)}, \\ \widehat{X}_i^{(r-1)}, & i \notin A^{(r)}. \end{cases} \tag{8}$$

Only active-token residuals are quantized, and inactive tokens require no new payload. Overall, token sparsity determines which tokens to update, quantization compresses them, and column sparsity determines which keys and values are read.

E Detailed Experimental Settings

Sparsity Setting In our experiments, we use the same configuration of FlashLoop for each model on all benchmarks. Table 5 summarizes the complete loop-wise sparsity setting used in our experiments.

Quantization Setting For both the KV base states and residuals, we use the same quantization scheme. Specifically, we apply INT4 asymmetric group-wise quantization, where the scaling factor and offset are computed from the minimum and maximum values within each group. The group size is set to 64. Following prior work (Liu et al., 2024b), we quantize post-RoPE keys using per-channel quantization, while values are quantized per token. In addition, we keep the most recent 64 tokens in BF16. In sparse loops, the K/V states generated by active tokens are packed as cross-loop residuals and then quantized using the same scheme described above.

Speedup and KV Cache Memory For speedup and KV cache memory measurement, we record the end-to-end wall clock time and peak KV cache memory for the whole benchmark experiments.

Hyperparameter We randomly sample 128 sequences from WikiText-2 as a fixed calibration set. For each model, we select the smallest retention ratio such that the relative reconstruction error of the corresponding hidden states or attention outputs remains below 5%. The resulting loop-wise sparsity schedule is calibrated once per model and then fixed across all downstream tasks and benchmarks.

F Long-Context Quality Evaluation

To evaluate the effect of FlashLoop in long-context setting, we compare the perplexity of Ouro-1.4B with and without FlashLoop in WikiText-2 (Merity et al., 2016) in context length of 8K, 16K and 32K tokens. Specifically, we construct a deterministic prefix from the same token stream and evaluate perplexity over the subsequent 128 ground-truth tokens. Moreover, we also use Needle-in-a-Haystack benchmark to evaluate the long context ability. For FlashLoop we use the same hyperparameters as main experiment. As illustrated in Table 6, across different long-context setting, FlashLoop does not bring significant performance degradation.

Table 5: **Loop-wise sparsity configurations used in the main experiments.** Token retention is the fraction of prompt-token rows recomputed during prefill. Column retention is the fraction of attention columns recomputed during decode.

Model family	Loop(s)	Tokens	Columns	Mode
Ouro-1.4B	1–2	100%	100%	Dense warm-up
	3	25%	10%	Sparse
	4	10%	10%	Sparse
Ouro-1.4B-Thinking	1–2	100%	100%	Dense warm-up
	3	25%	12%	Sparse
	4	10%	12%	Sparse
Ouro-2.6B	1–2	100%	100%	Dense warm-up
	3	20%	8%	Sparse
	4	8%	8%	Sparse
Ouro-2.6B-Thinking	1–2	100%	100%	Dense warm-up
	3	20%	8%	Sparse
	4	8%	8%	Sparse
Huginn-3.5B	1–4	100%	100%	Dense warm-up
	5–32	$90\% \rightarrow 20\%$	$76\% \rightarrow 10\%$	Sparse with smooth linear decay

Table 6: **Long-context quality evaluation.** WikiText-2 perplexity is measured on 128 continuation tokens (lower is better). Needle-in-a-Haystack accuracy is measured over full single-needle and multi-needles samples.

Setting	WikiText-2 PPL ↓			Needle-in-a-Haystack (%) ↑	
	8K	16K	32K	Single-needle	Multi-needles
Ouro-1.4B	10.526	**4.231**	**5.321**	83.0	**9.2**
Ouro-1.4B + FlashLoop	**10.513**	4.266	5.589	**84.5**	7.5

G Efficient GPU Implementation

We implement both a PyTorch reference implementation and an optimized GPU inference engine. We design the latter to translate FlashLoop's theoretical computation and memory reductions into actual runtime and memory savings, avoiding the kernel-launch, memory-traffic, and intermediate-tensor overheads of a direct PyTorch implementation.

- **Sparse execution over active tokens.** In later loops, we process only selected token rows through Transformer blocks, while inactive rows directly reuse hidden states and KV entries from the preceding loop.
- **Fused projections.** We compute Q, K, and V projections with a single GEMM and similarly fuse the gated MLP's gate and up projections, reducing kernel-launch overhead.
- **Physically packed KV cache.** We store KV states directly in a packed 4-bit representation on the GPU, together with lightweight quantization metadata and only a short full-precision residual tail.
- **Attention over compressed cross-loop KV.** For attention over quantized cross-loop KV, we use custom CUDA kernels to read packed anchor and residual KV streams directly during QK and PV computation, avoiding explicit reconstruction of a full BF16 KV cache.
- **Sparse decode kernels.** During late-loop decoding, FlashLoop executes attention only on selected key columns and corresponding value rows. We implement a custom kernel to fuse selection and computation, so that intermediate computation scales with the retained budget.

The engine is specialized for inference with BF16 activations on NVIDIA GPUs. It is intended as a reference implementation of FlashLoop's sparse execution and KV compression techniques. For efficiency evaluation of baseline, we also use inference engine but close KV cache quantization and sparsity configuration, which helps us to fairly evaluate the gain of our method.