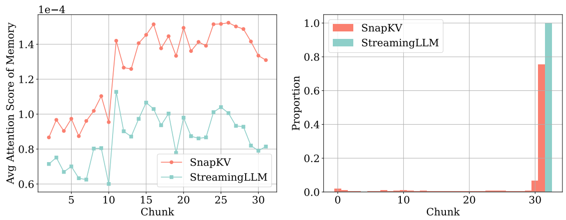
Memorize Step by Step: Efficient Long-Context Prefilling with Incremental Memory and Decremental Chunk

Anonymous ACL submission

Abstract

The evolution of Large Language Models (LLMs) has led to significant advancements, with models like Claude and Gemini capable of processing contexts up to 1 million tokens. However, efficiently handling long sequences remains challenging, particularly during the prefilling stage when input lengths exceed GPU memory capacity. Traditional methods often segment sequence into chunks and compress them iteratively with fixed-size memory. However, our empirical analysis shows that the fixed-size memory results in wasted computational and GPU memory resources. Therefore, we introduces Incremental Memory (IM), a method that starts with a small memory size and gradually increases it, optimizing computational efficiency. Additionally, we propose Decremental Chunk based on Incremental Memory (IMDC), which reduces chunk size while increasing memory size, ensuring stable and lower GPU memory usage. Our experiments demonstrate that IMDC is consistently faster (1.45x) and reduces GPU memory consumption by 23.3% compared to fixed-size memory, achieving comparable performance on the LongBench Benchmark.



(a) Memory Attention (b) Memory Distribution

Figure 1: (a): The average attention scores of memory at each step. (b): The distribution of memory content across chunks, where we count the number of key-value pairs in memory originating from each chunk. For both Figure (a) and (b), we used KV Cache pruner (SnapKV (Li et al., 2024) and StreamingLLM (Xiao et al., 2023)) to compress memory and chunk.

1 Introduction

The evolution of Large Language Models (LLMs) has reached new frontiers, with models like Claude (Anthropic, 2024) and Gemini (Reid et al., 2024) capable of processing contexts spanning up to a 1 million tokens. However, the efficiency of processing long sequences with LLM still faces significant challenges.

The inference of LLM can be divided into two parts: Prefilling and Decoding. LLM inference for long documents faces significant challenges in both stages. In the prefill stage, the model needs to read long sequences and endure the quadratic complexity of attention calculations with respect to the sequence length. During the decoding stage, decoding each token requires accessing the substantial Key-Value (KV) Cache generated in the prefill stage. Most efforts to optimize the efficiency of LLM for long sequence focus on the decoding stage, particularly on compressing the KV Cache (Xiao et al., 2023; Zhang et al., 2023; Liu et al., 2024c; Hooper et al., 2024; Liu et al., 2024a; Sun et al., 2024). However, when the input length during the prefilling stage exceeds the maximum length supported by GPU memory capacity, even prefilling cannot proceed. Existing works (Bulatov et al., 2023a,b; Ge et al., 2023b; Liu et al., 2020; Munkhdalai et al., 2024) tackle this problem by dividing the sequence into chunks and iteratively compress these chunks with a fixed-size buffer as memory.

Our anlysis on the memory displayed in Figure 1 reveals that: 1) the attention scores of memory starts at a relatively low value and gradually increases throughout the prefill process (Figure 1a.), which suggests that early-stage memory has minimal influence on the next-step computation; 2) once the prefill phase concludes, the memory distribution is primarily concentrated at the end of the sequence (Figure 1b), implying that most of the early-stage memory is not retained by the end of the prefill.

Overall, our finding suggest that the early-stage memory in the prefill phase is less impactful compared to the later-stage memory. Therefore, it is unnecessary to maintain a large memory size at the early stage of prefilling. This implies that approaches (Bulatov et al., 2023a,b; Ge et al., 2023b; Munkhdalai et al., 2024) that maintain a fixed-size buffer to compress long sequences may result in wasted computational and memory resources.

To avoid computational waste during the early stage of prefilling, we propose **Incremental Memory** (IM), which starts with a small memory size and gradually increases it until the end of the prefilling phase. During this growth phase, the memory size of IM remains smaller than the maximum length, resulting in greater efficiency compared to the commonly used fixed-size memory.

While analyzing memory distribution across different layers[1], we observed that higher layers exhibit a more uniform memory distribution compared to lower layers. Consequently, we propose an adaptive memory growth strategy to set memory sizes for each layer based on the proportion of memory retained after compression, with layers retaining more memory being allocated larger memory sizes.

Although IM is faster than fixed-size memory, it does not significantly reduce peak GPU memory usage, as the memory size of IM is the same as that of fixed-size memory at the end of the prefilling phase. Therefore, we propose **Decremental Chunk** based on **Incremental Memory** (IMDC), which starts with a large chunk size that decreases as memory size increases. When the memory size is small, the chunk size is large, and vice versa. The incremental memory and decremental chunk strategies complement each other, maintaining stable GPU memory usage that is lower than fixed-size memory, which is illustrated in Figure 2.

Our experiments show that IMDC is consistently faster (1.45x) than fixed-size memory and consumes less GPU memory (23.3% reduction) during the prefill stage, yielding comparable results on LongBench Benchmark (Bai et al., 2023).

Our main contributions are as follows:

- Our analysis on memory reveals that, the early-stage memory in the prefilling is less impactful than the later-stage memory.

[1]The results are shown in Figure 6

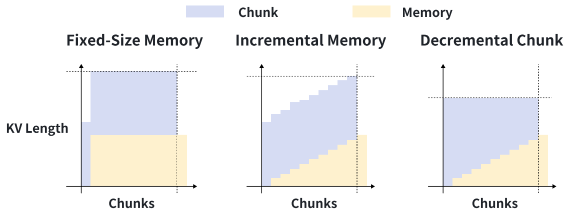


Figure 2: The illustration of Fixed-Size Memory, Incremental Memory (IM) and Incremental Memory with Decremental Chunk (IMDC).

- Based on this finding, we propose the Incremental Memory and Decremental Chunk (IMDC) approach, which dynamically increases memory size while decreasing chunk size.
- Our experiments demonstrate that IMDC is 1.45 times faster than the commonly used fixed-size memory and consumes 23.3% less GPU memory during the prefill stage, without sacrificing performance on long-context benchmarks.

2 Related Works

The long-context efficiency of LLM has been widely studied, which can be classified into two categories: prefilling and decoding.

Prefilling The prefilling of LLM encounters quadratic complexity in attention calculations with respect to sequence length. Numerous research efforts have sought to reduce this quadratic complexity through methods such as low-rank approximation (Wang et al., 2020; Peng et al., 2021; Choromanski et al., 2020) and sparsification (Child et al., 2019; Vyas et al., 2020; Kitaev et al., 2020). Tay et al. (2023) provided a comprehensive review of these approaches. These methods modify the computation mode of attention, often resulting in a trade-off with model performance. In contrast, flash attention (Dao et al., 2022) identified that the efficiency bottleneck lies primarily in input/output (I/O) operations rather than computational processes. By implementing CUDA operations, they significantly accelerated attention calculations without altering the fundamental computation of attention. RMT (Bulatov et al., 2023a) proposed an iterative compression scheme for long texts, maintaining and dynamically updating a fixed-size memory, which is followed by (Bulatov et al., 2023b; Ge et al., 2023b; Liu et al., 2020; Munkhdalai et al., 2024). AutoCompressors (Liu

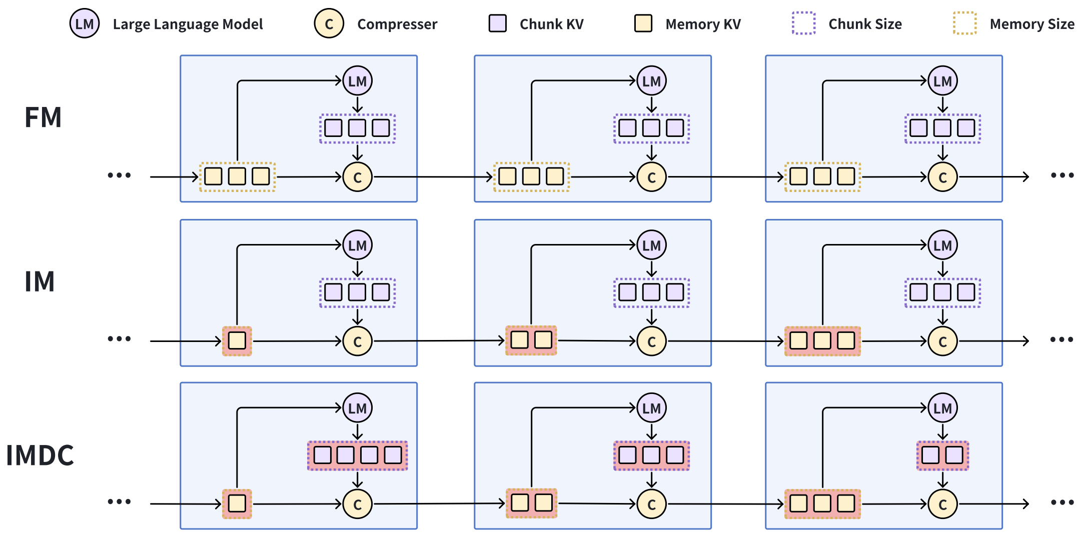


Figure 3: The illustration of iterative compression with Fixed-Size Memory (FM), Incremental Memory (IM) and Decremental Chunk based on Incremental Memory (IMDC). The iterative compression involves multiple steps of compression on the KV cache of memory and chunk.

et al., 2020) also introduced incremental memory, but different from our method, they increase memory size to enhance the model performance, which results in significant overhead. We demonstrate the superiority of our method compared to AutoCompressors empirically in Appendix B.2.

Decoding Most efforts to optimize the efficiency of long-context decoding have focused on KV Cache compression. Research in this area can be categorized into KV Cache Pruning (Zhang et al., 2023; Xiao et al., 2023; Liu et al., 2023), low-rank approximation (Shazeer, 2019; Ainslie et al., 2023; Shao et al., 2024), quantization (Liu et al., 2024c; Hooper et al., 2024; Liu et al., 2024b), and layer sharing (Liu et al., 2024a; Sun et al., 2024; Brandon et al., 2024). Key works in KV Cache pruning include H2O (Zhang et al., 2023) and StreamingLLM (Xiao et al., 2023). H2O selects important KVs based on cumulative attention scores, while StreamingLLM retains only the KVs closest to the end of the sequence. Subsequent works (Oren et al., 2024; Ge et al., 2023a; Dong et al., 2024; Ren and Zhu, 2024; Li et al., 2024) proposed several improvements to H2O, all of which determine KV importance based on attention scores. Notable approaches for low-rank approximation include multi-query attention (Shazeer, 2019) and grouped query attention (Ainslie et al., 2023), where different queries share the same KVs. Layer sharing methods (Liu et al., 2024a) identify redundancy among the KV Caches of different layers, retaining only the KVs of certain layers. Quantization compression (Liu et al., 2024c) reduces KV Cache precision from fp16 to int8 through various quantization methods (Dettmers et al., 2022).

In this paper, we adopted the iterative compression method from RMT. However, unlike RMT (Bulatov et al., 2023a), which compresses sequences into Soft Tokens, we used StreamingLLM and SnapKV to compress KV Cache, because they do not require training and can maintain a constant memory size during the iteration.

3 Method

3.1 Iterative Compression

When the input sequence length during the prefill stage exceeds the maximum length supported by the GPU memory limit, the sequence is segmented into multiple chunks and compressed iteratively, as illustrated in Figure 3. In each iteration, the LLM reads the memory as the KV cache for attention. After the attention computation, the newly generated KV cache is sent to the compressor, which updates the memory.

The process of iterating through chunks is similar to a recurrent neural network, while the computation within each chunk operates in parallel, akin to a transformer. [2]

[2]The intriguing intersection between KV Cache Pruning and recurrent neural networks is also discussed in Oren et al. (2024).

3.2 Incremental Memory

Based on the finding from Figure 1 that it is unnecessary to keep a large memory size at the early stage of prefilling, we propose Incremental Memory (IM), which increases memory size during the iteration of compression. We explore various incremental functions to increase memory size: Linear Function (Section 3.2), Adaptive Function (Section 3.2), and other increasing functions detailed in Appendix A.1.

Linear Function Suppose the number of chunks is n, the memory size increase from m_0 to $m_{\max}$ linearly:

$$m_i = \frac{(m_{\max} - m_0)i}{n-1} + m_0, \tag{1}$$

where n denotes the number of chunks. The middle section of Figure 3 illustrates the linear increase of memory size.

Adaptive Function By visualizing the memory distribution across layers [3], we observed significant differences in memory usage between high and low layers. Consequently, we propose Adaptive Function to allocate appropriate memory sizes for different layers. We record the memory retention ratio (the proportion of memory retained after the compression) of various layers. Suppose the memory of the j-th layer at the i-th step is $\mathbb{M}_i^j$, the memory retention ratio corresponding to that is defined as:

$$p_i^j = \frac{|\mathbb{M}_{i-1}^j \cap \mathbb{M}_i^j|}{|\mathbb{M}_i^j|}. \tag{2}$$

Intuitively, the more memory retained from the compression, the larger the memory size should be, and vice versa. Therefore, we can determine the memory size of each layer based on its memory retention ratio. We take the linear function as the basis, and scale it with the normalized memory retention ratio. Suppose that the number of layers is N, the memory size of the linear incremental memory of the j-th layer at the i-th step is b_i^j, then the memory size for adaptive incremental memory is:

$$m_i^j = \begin{cases} b_0^j & \text{if } i = 0 \\ \frac{p_j}{\sum p_j} N b_i^j & \text{if } i > 0 \end{cases} \tag{3}$$

[3]The visualization is shown in Figure 6

3.3 Decremental Chunk

Although incremental memory (IM) is faster than fixed-size memory, it does not significantly reduce peak GPU memory usage. To address this issue, we propose **Decremental Chunk** based on **Incremental Memory** (IMDC). IMDC begins with a large chunk size and decreases it as the memory size increases.

Regardless of changes in memory size and chunk size, IMDC maintains a constant average chunk size:

$$\frac{\sum_{i=0}^{n-1} c_i}{n} = c, \tag{4}$$

where c_i represents the chunk size at the i-th step, n is the number of chunks, and c denotes the average chunk size. Since the memory is not involved in the attention computation at the first step, the chunk size of IMDC at the first step is set to the average chunk size ($c_0 = c$).

At the i-th step, the attention key-value (KV) is the concatenation of the chunk at the i-th step and the memory at the $i-1$-th step. Therefore, the length of the attention KV at the i-th step is $c_i + m_{i-1}$. We set the chunk size to ensure that the attention KV length remains constant:

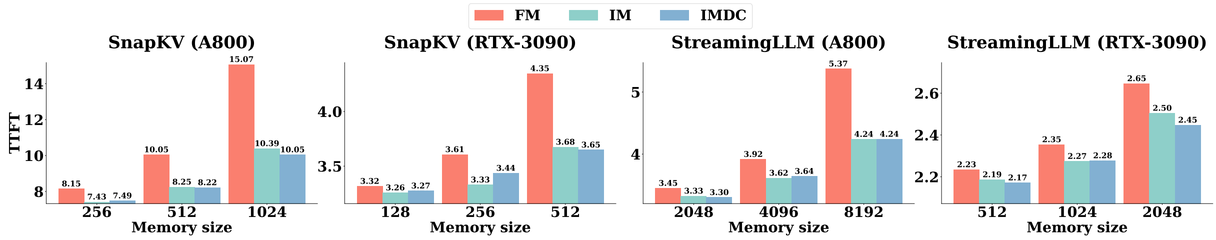
$$c_i + m_{i-1} = \frac{\sum_{i=1}^{n-1}(c + m_{i-1})}{n-1} \quad (i > 0), \tag{5}$$

where m_{i-1} is the memory size at the $i-1$-th step, and $\frac{\sum_{i=1}^{n-1}(c+m_{i-1})}{n-1}$ is the average length of the attention KV across all steps except the first step. Therefore, the chunk size of IMDC at the i-th step is:

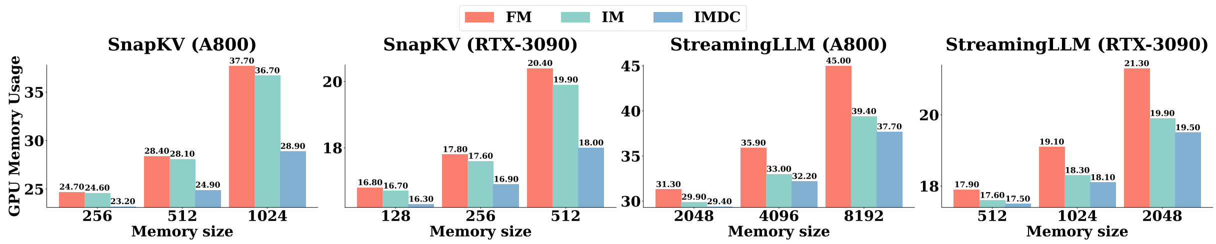
$$c_i = \begin{cases} c & \text{if } i = 0 \\ c + \hat{m} - m_{i-1} & \text{if } i > 0 \end{cases} \tag{6}$$

where $\hat{m} = \frac{\sum_{i=0}^{n-2} m_i}{n-1}$.

IMDC is illustrated on the bottom section of Figure 2, where the memory size increases while the chunk size decreases. When the memory size is small, the chunk size is large, and vice versa. The incremental memory and decremental chunk strategies complement each other, maintaining stable GPU memory usage. The attention KV length of IMDC remains constant at $c + \hat{m}$ (except for step 0), whereas for fixed-size memory it is $c + m_{\max}$. Since the memory size is incremental, we have $m_{\max} > \hat{m}$. Therefore, IMDC consumes less GPU memory than fixed-size memory.



(a) TTFT Comparison (Seconds). TTFT (Time To First Token) refers to the time cost associated with the model encoding the input sequence.



(b) GPU Memory Usage Comparison (GB).

Figure 4: TTFT and GPU Memory Usage of LLama2-7B with Fixed-Size Memory (FM) vs. that with our methods (Incremental Memory (IM) and Incremental Memory with Decremetnal Chunk (IMDC)). The setting of sequence length and chunk size followes Table 3. We use different memory sizes for SnapKV and Streaming LLM, because SnapKV requires attention scores which does not support flash attention.

4 Experiments

4.1 Experiment Settings

Iterative Compression We divided the sequence into non-overlapping windows and encode position embedding for memory and chunk at each iteration independently instead of reusing the position embedding from the previous steps. Incremental Memory employs the linear increase, with the initial memory size defined as $\frac{m_{\max}}{n}$, where $m_{\max}$ is the maximum memory size and n is the number of chunks. Unless otherwise specified, the configurations of Incremental Memory adhere to this setup.

KV Cache Compression We tried two pruning algorithms: SnaKV (Li et al., 2024) and StreamingLLM (Xiao et al., 2023). SnaKV filters important key-value pairs based on attention scores, while StreamingLLM selects the most recent key-value pairs without relying on attention scores.

Models We compared our methods with Fixed-Size Memory, abbreviated as FM. Our methods are labeled as IM (Incremental Memory) and IMDC (Incremental Memory with Decremental Chunk). Our experiments were conducted on LLaMA-2-7B (Touvron et al., 2023), Tiny-LLaMA (Zhang et al., 2024) (1.1B), and InternLM2 (Cai et al., 2024) (7B). We used Dynamic NTK (bloc97, 2023) to extend the context length of LLama2-7b and Tiny-LLama. We used flash attention (Dao et al., 2022) to accelerate the attention calculation. However, SnapKV requires attention scores hence is not compatible with flash attention.

Evaluation We used Collie (Lv et al., 2023) to implement our methods and evaluate our methods on LongBench (Bai et al., 2023) with OpenCompass (Contributors, 2023). Our Perplexity evaluation used the data collected by Liu et al. (2020), which are sampled from the Github and Arxiv subsets of Redpajama (Computer, 2023).

4.2 Efficiency Improvement

We evaluated the efficiency of our methods (IM and IMDC) on both NVIDIA A800 and NVIDIA RTX 3090 GPUs. The setting of chunk size and sequence length is shown in Appendix B.1. The results are shown in Figure 4.

Time Efficiency We compared the time efficiency of our method versus FM in terms of the time to first token (TTFT), the results of which are shown in Figure 4a. We found that our IM and IMDC consistently demonstrates greater efficiency than FM, regardless of the pruners used and the devices employed. Furthermore, the efficiency gap between them widens as the memory size increases. It is because that the larger memory size has a larger impact on the computation time.

In the A800 experiments, IMDC achieved up to approximately 1.45x (SnapKV) and 1.26x (StreamingLLM) speedup over FM. In the RTX 3090 experiments, the speedup of IM was 1.2x (SnapKV) and 1.08x (StreamingLLM). Increasing the memory size would make the speedup more significant.

The acceleration of our methods on SnapKV is more significant than that on StreamingLLM. This is because that SnapKV cannot use flash attention, leading to a higher proportion of time spent on attention calculation. These empirical results align with our theoretical analysis in Appendix A.2: the acceleration of incremental memory is influenced by two factors—the memory size and the proportion of time spent on attention calculation relative to total computation time.

GPU Memory Efficiency We evaluated the peak memory usage during model prefilling. The results, presented in Figure 4b, indicate that both IM and IMDC consume less GPU memory compared to FM. As the memory size increases, our method saves even more memory compared to the FM.

In experiments conducted on the A800, the IMDC reduced GPU memory usage by up to 23.3% for SnapKV and 16.2% for StreamingLLM compared to FM. Similarly, on the RTX 3090, the reductions were 11% for SnapKV and 8% for StreamingLLM.

IM also conserves GPU memory usage because the chunk at the i-th step is concatenated with the memory produced at the $i-1$-th step for attention. Assuming the iteration involves n chunks (0, 1, ..., $n-1$), the peak GPU memory is determined by the memory size at the $(n-2)$-th step rather than the last step. However, the GPU memory reduction achieved by IM is not as significant as that achieved by IMDC, especially in the SnapkV experiment, where the number of chunks is large.

4.3 Perplexity Comparison

We compare the perplexity (PPL) of LLama2-7b when using different types of memory: FM, IM, and IMDC. The test data for perplexity is sampled from Redpajama and encompasses two domains (GitHub and ArXiv). The sequence length and chunk size configurations adhere to the A800 settings specified in Appendix B.1.

The results shown in Figure 5 indicate that there is no significant difference in perplexity between IM/IMDC and FM for either SnapKV or

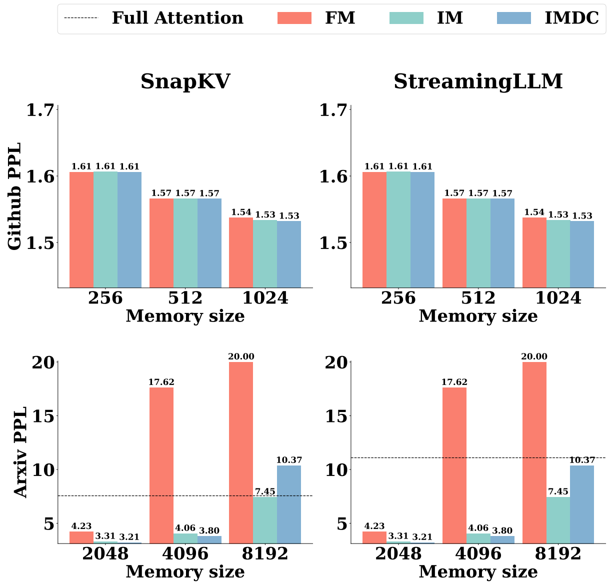


Figure 5: Perplexity of LLama2-7B with Fixed-Size Memory (FM) versus that with our methods (Incremental Memory (IM) and Incremental Memory with Decremental Chunk (IMDC)).

StreamingLLM. When the memory size is 1024, IM/IMDC even performs slightly better than FM. This may be because IM/IMDC selects KV pairs more concentrated towards the end of the sequence, which is beneficial for lowering PPL.

When the chunk size is 1024, a larger memory size results in a lower perplexity. However, when the chunk size is increased to 8192, the trend reverses, with a larger memory size leading to a higher perplexity. This is because 8192 exceeds the model's maximum supported length, and even with Dynamic NTK (bloc97, 2023), the PPL for Full Attention is high (GitHub PPL: 7.56, Arxiv PPL: 11.09). IM and IMDC consistently achieve lower PPL than FM and Full Attention. This is because the memory length of IM and IMDC increases gradually.

When comparing SnapKV and StreamingLLM, we observe that SnapKV achieves significantly lower perplexity than StreamingLLM under identical conditions.

4.4 Benchmark Comparison

We compared the performance of our methods (IM and IMDC) versus FM on LongBench. As shown in Table 1, the performance differences between IM and FM are minimal (<=0.5) under any settings. In most experiments, the performance differences between IM and FM are within 0.15. On InternLM2 and Tiny-LLaMA, IMDC is even better than FM.

Model	Pruner	Memory	Single-Doc QA	Multi-Doc QA	Summarization	Few-shot Learning	Synthetic	Code	Avg
LLaMA2-7b	Full-Attn	NA	16.4	7.89	11.61	50.58	3.68	63.34	28.15
	SnapKV	FM	15.63	**8.78**	11.83	48.17	3.50	**63.57**	27.85
		IM	15.53	8.75	11.74	**48.70**	4.41	63.51	**27.99**
		IMDC	**15.64**	8.47	**11.95**	46.78	**4.58**	63.32	27.65
	StreamingLLM	FM	12.89	7.90	**10.96**	**45.86**	3.40	**61.65**	**26.32**
		IM	**13.22**	7.92	10.90	44.47	3.86	61.44	26.14
		IMDC	12.95	**8.19**	10.78	44.88	**3.90**	61.23	26.15
InternLM2-7b	Full-Attn	NA	40.93	34.79	22.78	57.78	33.23	59.44	42.56
	SnapKV	FM	**23.50**	21.39	**17.88**	46.60	6.92	**59.87**	31.64
		IM	22.36	21.54	17.41	45.90	6.05	59.62	31.13
		IMDC	23.38	**22.38**	17.66	**48.67**	**8.45**	59.66	**32.24**
	StreamingLLM	FM	**23.14**	**21.49**	**16.79**	46.34	4.88	59.31	31.00
		IM	22.42	21.00	16.22	47.07	5.21	**59.95**	30.99
		IMDC	23.06	20.89	16.61	**47.31**	**5.73**	59.88	**31.25**
Tiny-LLaMA	Full-Attn	NA	2.77	0.99	5.76	2.12	0.59	18.06	5.78
	SnapKV	FM	16.06	9.43	16.91	**33.60**	2.95	50.27	23.50
		IM	16.22	9.41	15.74	31.09	2.85	**50.75**	22.98
		IMDC	**17.59**	**9.94**	**17.25**	33.16	**3.27**	49.58	**23.69**
	StreamingLLM	FM	16.31	10.07	16.77	31.46	2.96	51.92	23.60
		IM	16.32	9.85	17.07	30.37	**3.33**	51.81	23.45
		IMDC	**16.99**	**10.27**	**17.38**	**31.52**	2.41	**52.11**	**23.81**

Table 1: The performance comparison on LongBench. Full-attn: Full Attention; FM: Fixed-Size Memory; IM: Incremental Memory (ours); IMDC: Incremental Memory with Decremental Chunk (ours). For all models, both the chunk size and memory size are set to 1024.

This may be because the uneven Chunk Size is more closed to the full attention. An extreme case is that a sequence of length n is divided into two chunks with length $n-1$ and 1.

Overall, SnapKV performs better than StreamingLLM, especially on LLaMA2-7B. There is no difference in performance between SnapKV and StreamingLLM on Tiny-LLaMA, indicating that the attention scores of smaller models cannot reflect the importance of KV pairs.

We also found that the average score of Full Attention on Tiny-LLaMA is only 5.78, even with the Dynamic NTK. This is because the maximum sequence length that Tiny-LLaMA supports is limited to 2048 tokens. In contrast, the average scores of all iterative compression methods (FM, IM, and IMDC) exceed 20, indicating the superiority of iterative compression over full attention. We further compare the performance and efficiency between iterative compression and full attention in Appendix B.3.

Pruner	Incremental Function	Single-Doc QA	TTFT Time
SnapKV	LINEAR	22.35	5.11
	SQRT	**23.35**	5.83
	SQUARE	21.90	**4.49**
	SQUARE-SQRT	**23.24**	5.15
	ADAPTIVE	**23.20**	5.13
Streaming LLM	LINEAR	**22.41**	2.34
	SQRT	22.27	2.42
	SQUARE	22.08	2.27
	SQUARE-SQRT	21.97	2.35
	ADAPTIVE	**22.36**	2.36

Table 2: Performance comparison of different incremental functions. **LINEAR**: linear growth; **SQRT**: fast initial growth that slows down, in the form of $x^{1/2}$; **SQUARE**: slow initial growth that speeds up, in the form of x^2; **SQUARE-SQRT**: growth in the form of SQUARE in low layers and SQRT in high layers; **ADAPTIVE**: set the memory size based on the memory retention ratio in each layer. The specific formulas for the SQRT and SQUARE functions are described in Appendix A.1, and the implementation details of the ADAPTIVE function are provided in Section 3.2.We set both Chunk Size and Memory Size to 1024.

4.5 Optimal Incremental Strategy

In this experiment, we explored different functions to increase memory size and compared their impact on the performance and efficiency of the InternLM2-7b. We used InternLM2-7b for evaluation because the performance gap between IM/IMDC and FM on InternLM2-7b is more significant than that on LLama-7b.

The results are shown in Table 2. The outcomes for SnapKV matched our expectations. The SQRT function achieved the best performance, significantly outperforming the LINEAR function, but it was also the slowest among the five functions. This is reasonable because the memory size of the SQRT function is larger than that of the other func-

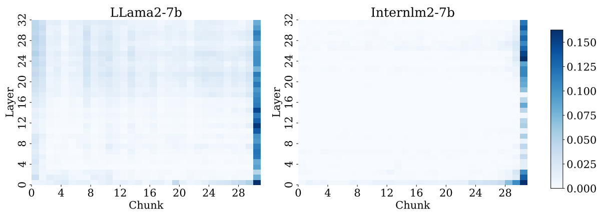


Figure 6: The memory distribution across different chunks in various layers for LLaMA2-7b and Internlm2-7b. The horizontal axis represents the Chunk ID, while the vertical axis represents the Layer ID. The intensity of the color reflects the proportion of memory distribution, with brighter colors indicating a higher proportion of memory within a given chunk. We have excluded the last column, as the majority of memory key-value pairs are concentrated in the final chunk.

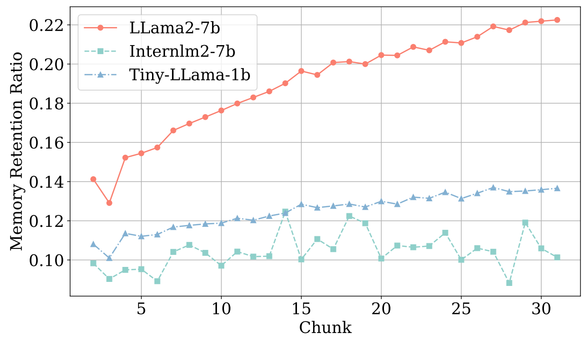


Figure 7: The variation of the Memory Retention Ratio during the iteration. The Memory Retention Ratio is defined as the proportion of memory retained after compression, see Equation 2. The higher Memory Retention Ratio indicates the less memory being forgotten after compression. The pruner used is SnapKV.

tions. Both SQUARE-SQRT and ADAPTIVE are designed to set the appropriate memory size for different layers. They exhibited the same performance as the SQRT function and the same efficiency as the LINEAR function. The SQUARE function was the most efficient among the five functions, but its performance was the worst.

As for the experiments on StreamingLLM, the impact of incremental functions was minimal. LINEAR and ADAPTIVE functions achieved the best performance. Overall, considering both performance and efficiency, the ADAPTIVE function is the optimal incremental function.

4.6 Analysis

Visualization of Memory Map We conducted a statistical analysis of the memory distribution across chunks by recording the chunk ID of each key-value pair in the memory. The data for this evaluation is the same as that used for the PPL Comparison (Section 4.3). For this analysis, we utilized fixed-size memory instead of incremental memory. The results, illustrated in Figure 1, indicate that the majority of the memory is concentrated in the last few chunks, irrespective of the models or pruners used.

We further investigated memory distribution across different layers for both LLaMA2-7b and InternLM2-7b. The results are shown in Figure 6. We found significant variation in memory distribution across different layers of LLaMA2-7b, with higher layers exhibiting a more uniform distribution than lower layers. Conversely, for InternLM2-7b, the differences in memory distribution across layers are minimal. Inspired by this observation, we propose adaptive Incremental Memory in Section 3.2.

Incremental Long-Term Memory The test data was sampled from the Github and Arxiv subsets of RedPajama, with each sample containing 32k tokens. During the iteration, the memory is continuously updating. We visualized the memory retention ratio defined in Equation 2 in Figure 7.

We observed that the memory retention ratio for LLaMA2-7b and Tiny-LLaMA increases linearly with iterations, whereas the memory retention rate for Internlm2-7b exhibits fluctuations. The increasing memory retention ratio suggests that as the model undergoes more iterations, it tends to retain more long-term memory.

5 Conclusion

In this paper, we addressed the inefficiencies in long-context prefilling of LLM by introducing two novel techniques: Incremental Memory and Decremental Chunk. Incremental Memory optimizes memory usage by dynamically increasing the memory size during prefilling, avoiding unnecessary computational overhead. Decremental Chunk complements this approach by dynamically adjusting the chunk size, maintaining stable and lower GPU memory usage. Our experimental results show that the combination of these methods significantly improves efficiency, with less prefilling times and GPU memory consumption compared to traditional fixed-size memory approaches.

Limitations

1. In our experiments, we tested the performance and efficiency of our methods using sequences with length of 32k tokens. However, iterative compression can support inputs of unlimited length. We have not yet validated the effectiveness of our method on longer sequences, such as those with one million tokens.
2. We have evaluated our methods on LLama2-7b, InternLM2-7b, and Tiny-LLama. However, due to limitations in computational resources, we have not tested our model on the larger models, such as LLama2-70b. Nevertheless, we believe our method is more suitable for larger models because the memory bottleneck is more pronounced in these cases.

Ethics Statement

This paper honors the EMNLP Code of Ethics. The dataset used in the paper does not contain any private information. The code will be are open-sourced under the MIT license.

References

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 4895–4901. Association for Computational Linguistics.

AI Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. *Claude-3 Model Card*.

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. *CoRR*, abs/2308.14508.

bloc97. 2023. Dynamically scaled rope further increases performance of long context llama with zero fine-tuning.

William Brandon, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan Kelly. 2024. Reducing transformer key-value cache size with cross-layer attention. *arXiv preprint arXiv:2405.12981*.

Aydar Bulatov, Yuri Kuratov, and Mikhail S. Burtsev. 2023a. Scaling transformer to 1m tokens and beyond with RMT. *CoRR*, abs/2304.11062.

Aydar Bulatov, Yuri Kuratov, and Mikhail S. Burtsev. 2023b. Scaling transformer to 1m tokens and beyond with RMT. *CoRR*, abs/2304.11062.

Zheng Cai, Maosong Cao, Haojiong Chen, Kai Chen, Keyu Chen, Xin Chen, Xun Chen, Zehui Chen, Zhi Chen, Pei Chu, Xiaoyi Dong, Haodong Duan, Qi Fan, Zhaoye Fei, Yang Gao, Jiaye Ge, Chenya Gu, Yuzhe Gu, Tao Gui, Aijia Guo, Qipeng Guo, Conghui He, Yingfan Hu, Ting Huang, Tao Jiang, Penglong Jiao, Zhenjiang Jin, Zhikai Lei, Jiaxing Li, Jingwen Li, Linyang Li, Shuaibin Li, Wei Li, Yining Li, Hongwei Liu, Jiangning Liu, Jiawei Hong, Kaiwen Liu, Kuikun Liu, Xiaoran Liu, Chengqi Lv, Haijun Lv, Kai Lv, Li Ma, Runyuan Ma, Zerun Ma, Wenchang Ning, Linke Ouyang, Jiantao Qiu, Yuan Qu, Fukai Shang, Yunfan Shao, Demin Song, Zifan Song, Zhihao Sui, Peng Sun, Yu Sun, Huanze Tang, Bin Wang, Guoteng Wang, Jiaqi Wang, Jiayu Wang, Rui Wang, Yudong Wang, Ziyi Wang, Xingjian Wei, Qizhen Weng, Fan Wu, Yingtong Xiong, and et al. 2024. Internlm2 technical report. *CoRR*, abs/2403.17297.

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

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Jared Davis, Tamás Sarlós, David Belanger, Lucy J. Colwell, and Adrian Weller. 2020. Masked language modeling for proteins via linearly scalable long-context transformers. *CoRR*, abs/2006.03555.

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

OpenCompass Contributors. 2023. Opencompass: A universal evaluation platform for foundation models. `https://github.com/open-compass/opencompass`.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Llm.int8(): 8-bit matrix multiplication for transformers at scale. *CoRR*, abs/2208.07339.

Harry Dong, Xinyu Yang, Zhenyu Zhang, Zhangyang Wang, Yuejie Chi, and Beidi Chen. 2024. Get more with LESS: synthesizing recurrence with KV cache compression for efficient LLM inference. *CoRR*, abs/2402.09398.

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2023a. Model tells you what to discard: Adaptive KV cache compression for llms. *CoRR*, abs/2310.01801.

Tao Ge, Jing Hu, Xun Wang, Si-Qing Chen, and Furu Wei. 2023b. In-context autoencoder for context compression in a large language model. *CoRR*, abs/2307.06945.

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. *CoRR*, abs/2401.18079.

Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr F. Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. Snapkv: Llm knows what you are looking for before generation. *ArXiv*, abs/2404.14469.

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.

Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. 2020. Autocompress: An automatic DNN structured pruning framework for ultra-high compression rates. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 4876–4883. AAAI Press.

Ruikang Liu, Haoli Bai, Haokun Lin, Yuening Li, Han Gao, Zhengzhuo Xu, Lu Hou, Jun Yao, and Chun Yuan. 2024b. Intactkv: Improving large language model quantization by keeping pivot tokens intact. *CoRR*, abs/2403.01241.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

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

Kai Lv, Shuo Zhang, Tianle Gu, Shuhao Xing, Jiawei Hong, Keyu Chen, Xiaoran Liu, Yuqing Yang, Honglin Guo, Tengxiao Liu, Yu Sun, Qipeng Guo, Hang Yan, and Xipeng Qiu. 2023. Collie: Collaborative training of large language models in an efficient way. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023 - System Demonstrations, Singapore, December 6-10, 2023*, pages 527–542. Association for Computational Linguistics.

Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. 2024. Leave no context behind: Efficient infinite context transformers with infini-attention. *CoRR*, abs/2404.07143.

Matanel Oren, Michael Hassid, Yossi Adi, and Roy Schwartz. 2024. Transformers are multi-state rnns. *CoRR*, abs/2401.06104.

Hao Peng, Nikolaos Pappas, Dani Yogatama, Roy Schwartz, Noah A Smith, and Lingpeng Kong. 2021. Random feature attention. *arXiv preprint arXiv:2103.02143*.

Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy P. Lillicrap, Jean-Baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, Ioannis Antonoglou, Rohan Anil, Sebastian Borgeaud, Andrew M. Dai, Katie Millican, Ethan Dyer, Mia Glaese, Thibault Sottiaux, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, James Molloy, Jilin Chen, Michael Isard, Paul Barham, Tom Hennigan, Ross McIlroy, Melvin Johnson, Johan Schalkwyk, Eli Collins, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, Clemens Meyer, Gregory Thornton, Zhen Yang, Henryk Michalewski, Zaheer Abbas, Nathan Schucher, Ankesh Anand, Richard Ives, James Keeling, Karel Lenc, Salem Haykal, Siamak Shakeri, Pranav Shyam, Aakanksha Chowdhery, Roman Ring, Stephen Spencer, Eren Sezener, and et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *CoRR*, abs/2403.05530.

Siyu Ren and Kenny Q. Zhu. 2024. On the efficacy of eviction policy for key-value constrained generative language model inference. *CoRR*, abs/2402.06262.

Zhihong Shao, Damai Dai, Daya Guo, and Bo Liu. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model.

Noam Shazeer. 2019. Fast transformer decoding: One write-head is all you need. *CoRR*, abs/1911.02150.

Yutao Sun, Li Dong, Yi Zhu, Shaohan Huang, Wenhui Wang, Shuming Ma, Quanlu Zhang, Jianyong Wang, and Furu Wei. 2024. You only cache once: Decoder-decoder architectures for language models. *arXiv preprint arXiv:2405.05254*.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2023. Efficient transformers: A survey. *ACM Comput. Surv.*, 55(6):109:1–109:28.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Apoorv Vyas, Angelos Katharopoulos, and François Fleuret. 2020. Fast transformers with clustered attention. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. 2020. Linformer: Self-attention with linear complexity. *CoRR*, abs/2006.04768.

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

Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. Tinyllama: An open-source small language model. *CoRR*, abs/2401.02385.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark W. Barrett, Zhangyang Wang, and Beidi Chen. 2023. H2O: heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

A Supplementary Method Details

A.1 Alternative Incremental Functions

We present several alternatives to the linear function for increasing memory size, namely the SQRT and the SQUARE:

$$m_i^{\text{square}} = \frac{(m_{\max} - m_0)i^2}{(n-1)^2} + m_0 \tag{7}$$

$$m_i^{\text{sqrt}} = \frac{(m_{\max} - m_0)\sqrt{i}}{\sqrt{n-1}} + m_0 \tag{8}$$

The growth rate of the SQRT is initially slow but accelerates over time, whereas the SQUARE function exhibits the opposite behavior. The memory size of the SQUARE function is smaller than that of the LINEAR, which in turn is smaller than that of the SQRT function.

Based on the memory distribution visualization in Section 4.6, we observed that the memory distribution in the higher layers of LLaMA2-7b is more uniform compared to the lower layers. Therefore, we propose a new increase function, SQUARE-SQRT, which combines the SQUARE and SQRT function: using SQUARE function for the lower layers, and SQRT function for the higher layers.

The integral of the sum of SQUARE and SQRT function ($m_i^{\text{high}} + m_i^{\text{low}}$) over the interval $[0, n-1]$ equals $n(m_{\max} + m_0)/2$, which is the same as that of linear function. Therefore, theoretically, the computational cost of SQUARE-SQRT is equivalent to that of LINEAR.

A.2 Time Complexity Analysis

The acceleration of IM over fixed-size Memory is determined by two factors: 1) the relative sizes of the memory size and chunk size; 2) the proportion of the total computation time occupied by the attention calculation. Assuming the maximum memory size is $m_{\max}$, the memory size at the i step is m_i, the chunk size is c, the number of chunks is n , then the acceleration of IM over fixed-size Memory is given by:

$$r\left(\frac{m_{\max} + c}{\hat{m} + c} - 1\right) + 1, \tag{9}$$

where $\hat{m} = \frac{\sum_{i=0}^{n-2} m_i}{n-1}$. Therefore, when $m_{\max} \gg c$ and r is close to 1, incremental memory achieves an ideal acceleration ratio: $\frac{m_{\max}}{\hat{m}}$.

Device	Pruner	Chunk Size	Sequence Length
A800	SnapKV	1024	32k
	StreamingLLM	8192	32k
RTX 3090	SnapKV	512	8k
	StreamingLLM	2048	8k

Table 3: Setting of Chunk Sizes and Sequence Lengths for Different Devices and Pruners

IM reduces the time complexity of the attention calculation from $O(ms + s^2)$ to $O(f(m, s) + s^2)$, where f depends on the specific incremental function. If f is a power function, the time complexity is $O(ms)$. If f is a reciprocal function, the time complexity is $O(\log(s)m + s^2)$.

B Supplementary Experiments

B.1 Experiment Setting of Chunk size and Sequence Length

Since the GPU memory of A800 is much larger than that of RTX 3090, we set a larger sequence length and chunk size for the experiment on A800. Furthermore, SnapKV does not support flash attention, hence the chunk size and sequence of which is larger than that of StreamingLLM. We report the detail setting in Table 3. Both the experiments of Efficiency Comparison (4.2) and PPL Comparison (4.3) follow this setting.

B.2 Incremental Fixed Memory Versus Incremental Dynamic Incremental

AutoCompressors (Liu et al., 2020) also dynamically increases the memory size while iterating over chunks. Although their memory size grows incrementally, they do not compress the existing memory; instead, they append the compressed chunks to the existing memory. In other words, their memory consists entirely of long-term memory that is neither updated nor forgotten. Conversely, our method updates the memory content through compression at each step.

Which kind of incremental memory is better? We compared the performance of them by evaluating the perplexity of LLaMA2-7b. The experimental setup is consistent with that in Section subsection 4.3, and the results are shown in Figure Figure 8. We refer to AutoCompressors (Liu et al., 2020) as Incremental Fixed Memory, and our method as Incremental Dynamci Memory.

According to Figure 8, the perplexity of Dynamic Incremental Memory is significantly lower

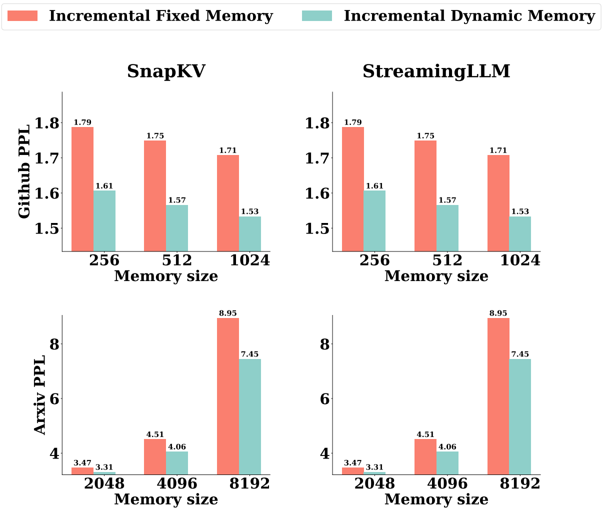


Figure 8: Incremental Fixed Memory (Liu et al., 2020) versus Incremental Dynamic Memory (ours). The data for evaluation is the same as that used in PPL Comparison (Section 4.3). Both approaches increase memory size linearly during iterative compression. For both methods, the chunk size and the maximum memory size are set to 1024.

GPU	Method	Save Logits	Max length
RTX 3090	Full Attention	Yes	8192
	Iterative Compression	Yes	65536
	Iterative Compression	No	infinity
A800	Full Attention	Yes	65536
	Iterative Compression	Yes	262144
	Iterative Compression	No	infinity

Table 4: The maximum input length supported by Full Attention and Interative Compression on A100 and RTX 3090 was evaluated. "Save Logits" refers to whether the model's output logits should be saved. We use IM for iterative compression which utilizes the StreamingLLM Pruner, both the chunk size and memory size of which are set to 1024.

than that of Fixed Incremental Memory in almost all configurations, which demonstrates the superiority of our method and suggests that memory needs to be updated, i.e., long-term memory alone is insufficient.

B.3 Why Iterative Compression?

To verify the advantages of iterative compression over Full Attention, we compared the maximum sequence length that iterative compression and Full Attention support at the prefilling stage. we set the memory size and chunk size to 1024 for IMDC and use the StreamingLLM pruner as the compresser.

The results are shown in the Table 4. Whether on the A800 or 3090, the maximum sequence

Method	Memory Size	Single-Doc QA	Time (seconds)
Iterative Compression	1024	22.36	3181.7
	2048	27.34	3187.2
	4096	34.88	3693.2
	8192	39.16	3802.9
Full Attention	NA	40.93	12101.4

Table 5: The performance and inference time of Full Attention Versus Iterative Compression with different memory sizes evaluated on a subset of LongBench (Single-Document QA). We use IM for iterative compression and LLama2-7b for the test model.

length supported by iterative compression is far greater than that supported by Full Attention (4 times greater). If we do not save model's logits, or only save the logits of the last chunk, iterative compression can support infinite sequence lengths.

In Table 1, the performance of InternLM2-7B with full attention is much better than iteractive compression (FM, IM, IMDC), particularly in QA tasks. We hypothesize that the small memory size is the primary cause of this discrepancy. Consequently, we conducted a comparative study of iterative compression and Full Attention with an increased Memory Size.

The results are shown in Table 5, where we can observe that increasing memory size is beneficial to narrow the gap between iterative compression and full attention. If the memory size is set to 8192, the performance of iterative compression on Single-Document QA is comparable with that of full attention, while requiring only 31% inference time.