Lil: Less is Less When Applying Post-Training Sparse-Attention Algorithms in Long-Decode Stage

Anonymous ACL submission

Abstract

Large language models (LLMs) demonstrate strong capabilities across a wide range of complex tasks and are increasingly deployed at scale, placing significant demands on inference efficiency. Prior work typically decomposes inference into prefill and decode stages, with the decode stage dominating total latency, especially in reasoning-intensive tasks. To reduce time and memory complexity in the decode stage, a line of work introduces sparse-attention algorithms. In this paper, we show, both empirically and theoretically, that sparse attention can paradoxically increase end-to-end complexity: information loss often induces significantly longer sequences, a problem that we term "Less is Less" (Lil). To mitigate the Lil problem, we propose an early-stopping algorithm that detects the threshold where information loss exceeds information gain during sparse decoding. Our early-stopping algorithm reduces token consumption by up to 90% with a marginal accuracy degradation of less than 2% across reasoning-intensive benchmarks.

1 Introduction

Large language models (LLMs) (OpenAI; Dai et al., 2024) exhibit strong capabilities across a wide range of complex tasks, such as automatic code generation (Wang et al., 2023) and creative writing (Bai et al., 2024a). Users interact with LLMs through natural-language-based prompts—sequences of tokens[1]. This combination of usability and capability has driven rapid and widespread adoption. As a result, LLMs must now be deployed at scale to handle an increasingly large and diverse set of requests, including long-input requests (e.g., document-question answering (Bai et al., 2024b,a; Beltagy et al., 2020; Wu et al., 2024)), long-output requests (e.g., chain-of-thought reasoning (AIME; Hendrycks et al., 2021; Yao et al., 2023), long-form writing (Bai et al., 2024a), code generation (Wang et al., 2023)), as well as requests requiring both (Wu et al., 2025). Longer inputs and outputs significantly increase inference latency and resource consumption, posing substantial challenges for large-scale deployment.

To address these inference challenges, prior work typically decomposes inference into two stages: prefill and decode. In the **prefill** stage, the model processes tokens given by users. It computes the Key (K) and Value (V) vectors for all tokens, stores these vectors in the KV cache, and generates the first output token to initiate the decode stage. In the **decode** stage, the model iteratively processes each newly generated token. It computes the KV vectors for the new token, appends these vectors to the KV cache, and generates the next token. This process repeats until a specified stopping criterion is met. This paper focuses on accelerating the decode stage, which dominates total inference time (Hu et al., 2024; Yao et al., 2023), especially in reasoning-intensive tasks.

To optimize the decode stage, a major line of work introduces sparse-attention algorithms[2], aiming to reduce both time and memory complexity (Zhang et al., 2023; Xiao et al., 2024b; Tang et al., 2024; Hu et al., 2024; Chen et al., 2024). First, sparse attention reduces time complexity. Full attention requires each decode token to attend to all previous tokens. In contrast, sparse attention requires each decode token to attend to only the top-k most relevant tokens, substantially reducing computation while maintaining accuracy. Second, sparse attention may reduce memory complexity. Some algorithms reduce memory by discarding irrelevant KV vectors during decoding (Zhang et al.,

[1] Tokens may be thought of roughly as words.

[2] Although sparse attention also applies to prefill, this work focuses on the decode stage. We also emphasize post-training sparsity. Training-aware sparsity algorithms such as DeepSeek NSA (Native Sparse Architecture) (Yuan et al., 2025) fall outside our scope and are discussed in related work.

2023; Xiao et al., 2024b; Hu et al., 2024), while others retain the full KV cache (Tang et al., 2024; Chen et al., 2024), but therefore cannot reduce the memory footprint.

Although sparse attention algorithms appear beneficial, we find that they often increase **end-to-end** time and memory complexity due to frequent loss and recomputation of information, a problem that we term as Lil (Less-Is-Less)[3]. First, sparse attention increases **end-to-end** time complexity. While each decode step becomes faster, the information lost during sparse attention forces the model to generate longer sequences to compensate. Second, sparse attention increases **end-to-end** memory complexity. Although each step may store fewer KV vectors, the extended generation process increases memory residency time, negating potential savings.

This paper identifies and mitigates the Lil problem—the "elephant in the room" of sparse attention research—in three steps. First, through systematic empirical study, we demonstrate that widely used sparse-attention algorithms consistently increase output length by up to 90% on reasoning-intensive datasets compared with full attention. The outputs exhibit a clear pattern of information loss followed by attempted reconstruction (Section 3). Second, we further analyze the outputs through the lens of information theory (Section 4). We establish a quantitative relationship between the information of a sentence and its compression ratio, and further observe that, under sparse attention, the information of generated sequences does not necessarily increase as generation proceeds. Third, motivated by these empirical and theoretical findings, we propose *Guardian*, an early-stopping algorithm that halts decoding when the information of the generated sequence ceases to increase (Section 5). Such unnecessary continued generation arises in two cases: (i) the model fails to solve the task and generates indefinitely due to lost context, and (ii) the correct answer has already been produced, but the model continues verification and subsequently forgets that the answer has already been generated.

We implement a unified framework that supports the integration and comparative evaluation of diverse sparse-attention algorithms. We also incorporate *Guardian* into this framework. Our evaluation yields two key findings (Section 6). First, on reasoning-intensive benchmarks (Hendrycks et al., 2021; AIME; Cobbe et al., 2021), *Guardian* reduces total token wastage by up to 90% compared to decoding without early stopping, with less than 2% accuracy drop. Second, experiments show that *Guardian* can also be applied to general cases of prolonged Chain-of-Thought (CoT) generation even without sparse-attention algorithms[4].

In this paper, we make the following three main contributions:

- We identify and characterize the Lil problem in existing sparse-attention algorithms through a systematic empirical study.
- We establish a connection between the information of a sentence and its compression ratio using entropy-based compression algorithms.
- We propose *Guardian*, an early-stopping algorithm (for the decode stage) that reduces token usage by up to 90% with less than 2% accuracy drop on reasoning-intensive benchmarks.

2 Background and Motivation

This section reviews the LLM inference pipeline and existing sparse-attention algorithms, and highlights the key challenges that motivate our study.

2.1 Autoregressive Generation and KV Cache

LLMs generate tokens autoregressively, involving two stages: the prefill stage and the decode stage. In the **prefill** stage, LLMs process the entire user-provided input (prompt or a sequence of tokens) $(x_1, x_2, \ldots, x_n)$. LLMs compute the Key (K) and Value (V) vectors for all tokens, store these vectors in the KV cache, and generate the first output token to initiate the decode stage. The prefill stage can be slow for long inputs, and the time to generate the first token is measured by the Time-to-First-Token (TTFT) metric. In the **decode** stage, LLMs generate one token at a time. The model computes the probability of the next token x_{n+1}, selects the most likely token, and appends its key and value vectors to the KV cache. This process repeats until a specified stopping criterion is met. The latency between consecutive tokens is measured by the Time-Between-Tokens (TBT) metric.

[3]Lil abbreviates "little" (dropping "tt" and 'e"). It also suggests that sparse-attention algorithms yield "little" benefit. Pronunciation resembles "Leo."

[4]Prolonged CoT arises from ill reasoning patterns induced by data quality issues, human preference biases for long sentences, or reward hacking, rather than from the information-loss-and-reconstruction characteristic of the Lil problem; despite related, it lies beyond the scope of this work.

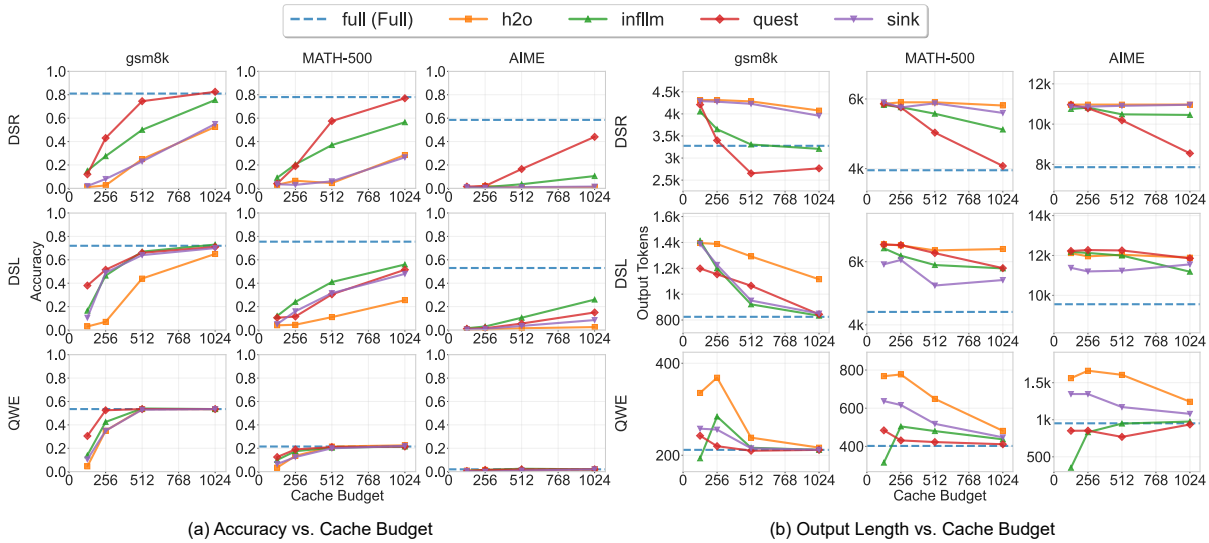

Figure 1: Accuracy/output length vs. cache budget for five algorithms (legends) across three datasets (rows) and three models (columns). DSR, DSL, and Qwe denote DeepScaleR-1.5B-Preview, DeepSeek-R1-Distill-Llama-8B, and Qwen1.5-MoE-A2.7B-Chat, respectively. The x-axis shows varying cache budgets. In (a), the y-axis shows the proportion of correctly solved problems over 200 test cases. In (b), the y-axis shows the average output length over the same 200 test cases. **For sparse-attention algorithms, the maximum generation length is capped at twice that of the full-attention baseline to prevent non-terminating generation.**

2.2 Cost of Long-Prefill and Long-Decode Inference

Both long-prefill and long-decode inference incur substantial memory and time overheads. In terms of **memory**, processing 128k tokens with the LLaMA 3.1 8B model in FP16 requires up to 16 GB, in addition to the 16 GB of model parameters[5]. In terms of **time**, inference on 32k tokens can take from tens to thousands of seconds on vLLM 0.6.1 with the same model (Hu et al., 2024), with large variance driven primarily by the decode stage: longer generations incur proportionally higher latency.

The decode stage dominates end-to-end inference time, especially for reasoning-intensive tasks (OpenAI; Wang et al., 2024; Zhao et al., 2024; Wei et al., 2022). For instance, the OpenAI o1 (OpenAI) may spend tens to hundreds of seconds in internal "thinking" before producing a final answer[6]. Accordingly, this paper focuses on optimizing long-decode inference.

2.3 Post-Training Sparse-Attention Algorithms for Long-Decode Optimization

To reduce memory and time complexity in LLM inference, a substantial body of work proposes sparse-attention algorithms (Xiao et al., 2024b; Zhang et al., 2023; Tang et al., 2024; Chen et al., 2024; Yuan et al., 2025), which restrict attention computation at each decode step to a small subset of critical tokens—often fewer than 10% of the context (Tang et al., 2024). First, sparse attention reduces time complexity. Full attention requires each decode token to attend to all previous tokens. In contrast, sparse attention requires each decode token to attend to only the top-k most relevant tokens. Second, sparse attention may reduce memory complexity. Some algorithms reduce memory by discarding irrelevant KV vectors during decoding, such as H_2O and Sink (Zhang et al., 2023; Xiao et al., 2024b; Hu et al., 2024), while others retain the full KV cache, such as infLLM and Quest (Tang et al., 2024; Chen et al., 2024; Xiao et al., 2024a), but therefore cannot reduce the memory footprint.

Sparse-attention algorithms can be broadly categorized into two types. **Training-aware** algorithms incorporate sparsity directly into the model architecture and training procedure. Despite being effective, they require architectural modifications and incur substantial training costs, and their sparsity is difficult to disable once deployed. Representative examples include DeepSeek Native Sparse Attention (NSA) and DeepSeek Sparse Attention (DSA) (Dai et al., 2024; Liu et al., 2024). In contrast, **post-training** algorithms apply sparsity to fully trained dense models at inference time. These

[5] https://huggingface.co/blog/llama31

[6] https://www.reddit.com/r/OpenAI/comments/1frdwqk/your_longest_thinking_time_gpt4_o1_o1mini/

algorithms are plug-and-play and training-free, and reportedly have demonstrated strong performance in preserving accuracy while reducing latency and memory consumption. This paper focuses on Post-Training Sparse-attention algorithms in the Decode stage (PTSD).

2.4 Limitations of PTSD

Although sparse attention algorithms appear beneficial, we find that they often increase end-to-end time and memory complexity due to frequent loss and recomputation of information. First, sparse attention increases end-to-end time complexity. Although sparsity reduces the per-step TBT, the loss of contextual information frequently forces the model to generate longer outputs to compensate. The resulting Job Completion Time (JCT),

$$\text{JCT} \uparrow = \text{TTFT} + \text{decode\_length}\uparrow \times \text{TBT} \downarrow, \quad (1)$$

increases (Section 3). Second, sparse attention increases end-to-end memory complexity. Although each decode step may store fewer KV vectors, the extended generation process increases memory residency time, negating potential savings.

We refer to the length-increasing problem as Lil (Less-Is-Less), which is the "elephant in the room" for the PTSD community. If left unaddressed, it undermines the fundamental motivation for adopting sparse-attention algorithms. In the subsequent sections, we first analyze the causes of this problem and then propose algorithms to mitigate it.

3 Empirical Study of Lil

We evaluate five sparse-attention algorithms across three datasets and three models (see Section 6 for detailed experiment setup), and obtain two key findings (Figure 1). First, accuracy increases with cache budget. H_2O and Sink are less accurate under fixed budgets because they discard KV vectors. Once important information is removed, the model cannot recover it. Quest and infLLM keep all KV vectors. They maintain higher accuracy but use more memory. Second, output length falls as the cache budget increases, but it remains longer than Full attention (by up to 90%). With small cache budgets, information loss outpaces information gain, and the model repeats content when trying to rebuild context (Figure 5 (a)). The model may fail to solve the task and generate indefinitely due to lost context. With larger cache budgets, the model makes partial progress and solves more

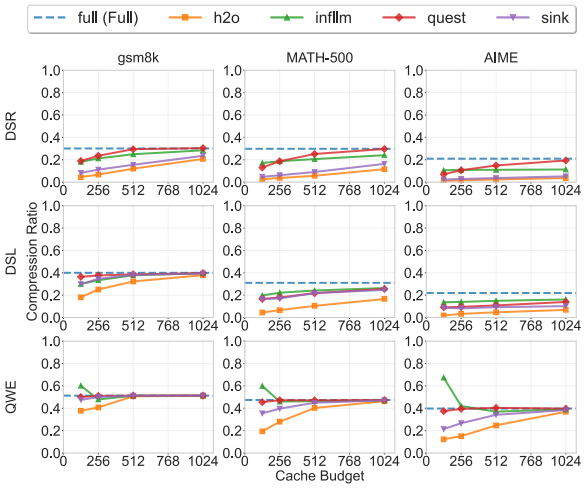

Figure 2: Compression ratio vs. cache budget. Notations follow Figure 1. The y-axis shows the average compression ratio (compressed-sequence length / original-sequence length) over 200 test cases.

cases; however, the outputs may still be excessively long, as the correct answer is produced early but the model continues verification and subsequently forgets that it has already generated the answer (Figure 5 (b)).

4 Compression Theory

Section 3 shows the Lil problem: information is lost under sparse attention, and models try to regain it by generating more tokens. We next use information theory to analyze this loss and gain.

To this end, we adopt LZ77 (Ziv and Lempel, 1977), a compression algorithm, which is simple, efficient, and grounded in theory. First, the key insight of LZ77 is simple: replace repeated substrings with concise references (e.g., offset-length pairs) to their earlier occurrences. When later segments of a sequence frequently recur in earlier segments, the resulting compression ratio (compressed length divided by original length) is small. Second, LZ77 is computationally efficient: compressing a sequence of 128k tokens takes approximately 34 ms (Section 6), which is comparable to the time to decode a single token (Zheng et al., 2024). Third, LZ77 admits strong theoretical guarantees. The achieved compression ratio ρ satisfies

$$\rho - \epsilon(L_s) \leq h(L_s - 1) \leq \rho. \quad (2)$$

where $h(k)$ denotes persymbol entropy, and $\epsilon(L_s) = \mathcal{O}(\log L_s / L_s)$. Consequently, ρ estimates information entropy up to a small term. Lower ρ indicates less new information and more redundancy. Please refer to the appendix for a comprehensive illustration of the LZ77 algorithm and related proof.

Algorithm 1 *Guardian* Algorithm

Input: A sequence X of prefill tokens, a model M, a frequency f, and a threshold t
Output: A sequence Y of prefill tokens plus decode tokens

```
cnt = 1
lastCompress = LZ77(X)
curCompress = lastCompress

Y = X
y = M.forward(Y, "prefill")
while y ≠ eos and len(Y) < M.context_len() and not is_early_stop(Y)
    Y.append(y)
    y = M.forward(Y, "decode")
Return Y

Function is_early_stop(Y)
if cnt % f == 0
    curCompress = LZ77(Y)
    if curCompress - lastCompress < t
        Return True
    lastCompress = curCompress
cnt = cnt + 1
Return False
End Function
```

We compress all sequences in Figure 1 and report their corresponding compression ratios in Figure 2, from which we draw two key insights. First, despite producing longer outputs, sparse-attention algorithms generate sequences with substantially less information than full attention. This result indicates that the model largely repeats earlier content to reconstruct lost information, leading LZ77 to encode much of the later sequence as references to previous segments. Second, as the cache budget increases, information gain increasingly outpaces information loss. As a result, the model attains higher accuracy with fewer tokens, diminishing the need for information reconstruction. Correspondingly, the compression ratio approaches that of full attention, reflecting more informative and less redundant generation.

5 Early-Stopping Algorithm

Based on the preceding empirical and theoretical analyses and Figure 3, we observe that under sparse-attention algorithms, the model may cease to gain new information while continuing to generate tokens. Motivated by this insight, we propose *Guardian*, an early-stopping algorithm that detects sustained stagnation in information gain during generation and terminates decoding early to reduce token consumption.

The algorithm is shown in Algorithm 1. First, in the prefill stage, the user prompt is processed and compressed to obtain *lastCompress*, the number of bytes left after compression. Second, the model enters the decode stage and stops upon generating an *eos* token or reaching the maximum context length. *Guardian* introduces an additional stopping condition: every f decode steps, the current sequence (prefill plus generated tokens) is compressed to obtain *curCompress* and compared with *lastCompress*. If the increase is below a threshold t bytes, we believe that the sequence has gained negligible new information—i.e., the newly generated tokens are largely redundant under LZ77—and stop the generation.

We next discuss the choice of f and t. First, the interval f must balance computational overhead and responsiveness: a small f incurs frequent compression and excessive overhead, whereas a large f reduces the sensitivity of *Guardian* and delays termination, thereby diminishing token savings. We set $f = 250$ in our experiments. In practice, f primarily affects token savings and end-to-end latency, but not accuracy, and can be adjusted based on model architectures, sequence lengths, and GPUs, which determine the per-step decoding cost. Second, given a fixed f, the threshold t is chosen such that t/f lies between the slopes of the initial growth stage and the subsequent plateau stage in Figure 3. Empirically, the slope in the plateau stage is below 0.02 across datasets and models, whereas the initial slope exceeds 1, and we set $t = 20$ such that $t/f \approx 0.08$. Because the transition between the two stages is sharp, *Guardian* is robust to variations in t as long as t/f falls within this range.

6 Evaluation

We begin by describing the experimental setup, including implementation details, datasets, models, evaluation metrics, and software/hardware environment. We then present key evaluation results.

6.1 Experimental Setup

Implementation. We implement *Guardian* based on Hugging Face (Hugging Face) with 2k lines of Python code. We port the LZ77 algorithm from the

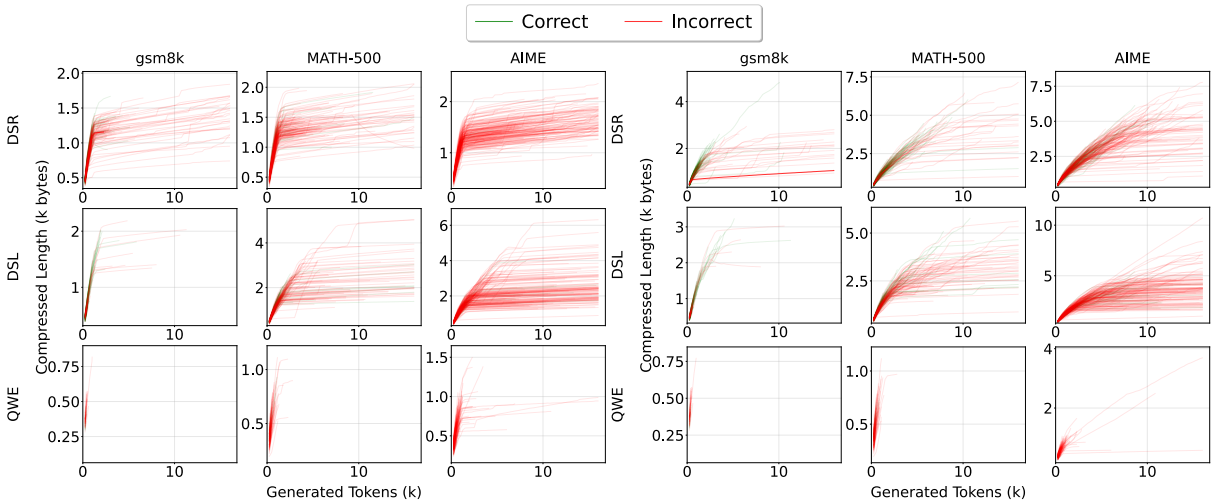

Figure 3: Average compressed length vs. original length for (a) sink (1024 cache budget) and (b) Quest (1024 cache budget) across three models and three datasets. Each line indicates an individual test case, with green and red curves denoting correct and incorrect outcomes, respectively. Other notations follow Figure 1. As the model generates more tokens (i.e., as the original length increases), the compressed length initially grows rapidly and then plateaus. Green curves denote correct test cases, whereas red curves denote incorrect test cases.

		GSM8K				MATH-500				AIME			
	%	H2O	Sink	Quest	infLLM	H2O	Sink	Quest	infLLM	H2O	Sink	Quest	infLLM
DSR	128	67.8/0.0	68.1/0.0	6.7/0.0	6.4/0.0	81.8/0.0	84.1/+1.5	11.5/0.0	8.3/0.0	93.7/0.0	92.5/0.0	18.4/-0.5	14.1/0.0
	256	62.1/0.0	64.9/+0.5	10.2/0.0	5.4/-0.5	78.0/-0.5	79.9/+1.0	11.2/0.0	9.0/-0.5	91.2/0.0	90.9/-0.5	19.7/-0.5	16.8/+0.5
	512	49.3/0.0	53.9/+1.0	6.1/-1.0	6.4/-0.5	66.8/0.0	70.9/-0.5	8.6/0.0	10.1/+0.5	86.4/0.0	87.8/-0.5	20.3/0.0	24.1/0.0
	1024	36.9/0.0	32.8/-1.5	9.8/-2.5	10.9/-1.5	53.6/0.0	51.8/-0.5	6.0/-0.5	11.9/0.0	79.6/0.0	79.9/0.0	14.5/-1.0	33.2/+0.5
DSL	128	22.7/0.0	8.1/0.0	1.6/0.0	0.4/0.0	75.5/0.0	47.9/+1.0	15.1/+1.0	3.8/+0.5	91.0/0.0	71.0/0.0	23.2/+0.5	4.8/0.0
	256	23.4/0.0	10.2/0.0	3.6/-0.5	0.7/0.0	75.8/0.0	45.8/-0.5	23.9/+2.5	1.7/0.0	87.9/0.0	72.6/0.0	44.9/0.0	3.9/0.0
	512	18.5/0.0	6.4/+1.0	2.4/+0.5	0.3/-0.5	71.0/+0.5	35.7/-1.0	26.3/-1.0	3.9/-0.5	87.4/0.0	67.6/0.0	53.9/0.0	10.0/-0.5
	1024	9.6/0.0	3.6/-0.5	1.5/-0.5	0.3/0.0	58.8/0.0	29.9/0.0	22.3/+2.0	5.9/0.0	83.6/0.0	65.0/0.0	48.2/-1.0	16.0/-2.0
Qwe	128	0.9/0.0	1.1/0.0	0.0/0.0	0.0/0.0	16.7/0.0	14.2/-0.5	0.2/0.0	0.0/0.0	39.5/0.0	36.8/+0.5	1.4/0.0	0.2/0.0
	256	0.5/0.0	1.7/0.0	0.0/0.0	0.0/0.0	19.3/0.0	13.7/0.0	0.0/0.0	0.4/0.0	42.7/0.0	31.7/0.0	1.1/0.0	0.8/0.0
	512	1.3/0.0	0.1/0.0	0.0/0.0	0.0/0.0	12.5/+0.5	6.0/0.0	0.1/0.0	0.4/0.0	31.6/0.0	16.1/0.0	0.6/0.0	1.1/0.0
	1024	0.2/0.0	0.0/0.0	0.0/0.0	0.0/0.0	3.2/0.0	0.8/0.0	0.2/0.0	0.6/0.0	11.0/0.0	6.0/0.0	1.6/+0.5	2.7/0.0

Table 1: Average token savings and accuracy degradation after applying *Guardian*. Notations follow Figure 1. Cell entries follow the format $Savings/\Delta Accuracy$, representing the percentage of relative token savings and the corresponding relative accuracy shift. All values are expressed in %. Please refer to Figure 1 for absolute token and accuracy numbers.

famous compression tool Gzip[7].

Datasets. We take the first 200 test cases from each of the following three open-source datasets for our benchmarks: GMS8K (Cobbe et al., 2021), MATH500 (Hendrycks et al., 2021), and AIME (AIME), to test the reasoning ability of language models. First, GMS8k (Cobbe et al., 2021) contains 8.5k high-quality, linguistically diverse grade-school math problems. These human-written problems need solutions that involve multi-step reasoning and a series of basic arithmetic operations. Second, MATH500 (Hendrycks et al., 2021) contains 500 challenging problems sourced from high school math competitions with five distinct levels based on the Art of Problem Solving (AoPS) framework, ranging from level 1 to level 5. Third, AIME (AIME) is a math problem dataset collected from the American Invitational Mathematics Examination (AIME) competition from 1983 to 2024, designed to challenge the most exceptional high school math students in the United States. These problems cover various fields, such as algebra, geometry, and number theory.

Models. We evaluate our algorithm using three popular models: DeepScaleR-1.5B-Preview[8], DeepSeek-R1-Distill-Llama-8B (DeepSeek-AI, 2025), and Qwen1.5-MoE-A2.7B-Chat[9]. These models span diverse architectures (dense vs. MoE),

[7]https://www.gzip.org/

[8]https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-Model-by-Scaling-RL-19681902c1468005bed8ca303013a4e2

[9]https://qwenlm.github.io/blog/qwen-moe/

training recipes, and parameter scales. In this paper, we use DSR to denote DeepScaleR-1.5B-Preview, DSL to denote DeepSeek-R1-Distill-Llama-8B, and Qwe to denote Qwen1.5-MoE-A2.7B-Chat.

Metrics. We evaluate efficiency and accuracy using two metrics. *Token savings* is defined as the ratio of the number of tokens generated without *Guardian* to that generated after applying *Guardian*. *Accuracy* (Wang et al., 2024) measures the mathematical equivalence between an LLMs output and the ground-truth answer. For each test case, it is either correct or incorrect, and the overall accuracy is reported as the percentage of correctly solved problems across the entire dataset. When reporting accuracy changes, we use the absolute difference in accuracy, expressed in percentage points.

Baselines. We compare Full, H_2O, StreamingLLM, InfLLM, and Quest—spanning sparse-attention algorithms from the earliest proposals to the state of the art as of October 2025—with and without *Guardian*. For each algorithm, the cache budget denotes the maximum number of tokens that a decoding token can attend to. For H_2O and Sink, the cache budget additionally specifies the number of tokens whose KVs are retained, as these algorithms attend to only preserved tokens. In contrast, infLLM and Quest retain the KVs of all tokens.

Environment. We run experiments on a single NVIDIA A100 server with one A100-80GB GPU available. It has a 128-core Intel(R) Xeon(R) Platinum 8358P CPU@2.60GHz with two hyperthreads and 1TB DRAM. We use Ubuntu 20.04 with Linux kernel 5.16.7 and CUDA 12.6.

6.2 Evaluation Results

***Guardian* saves up to 90% tokens with less than 2% accuracy drop.** Table 1 reports the end-to-end results after applying *Guardian*, from which we draw four key findings. First, *Guardian* reduces token usage by up to 90% while incurring less than a 2% accuracy drop, demonstrating its effectiveness. Second, in some cases, *Guardian* even improves accuracy. This improvement occurs when the model generates the correct answer early but continues decoding, redundantly re-evaluating the solution, and ultimately loses the correct answer. Early termination prevents such degradation. Third, as the sparsity algorithm becomes more conservative (e.g., from Sink to Quest, or from a cache budget of 128 to 1024), token redundancy decreases, resulting in fewer tokens saved. Nevertheless, even the most conservative sparse configurations that achieve accuracy comparable to full attention still exhibit measurable redundancy and benefit from early stopping (e.g., the DSR model on GSM8K with Quest and a 1024 cache budget). Fourth, *Guardian* yields negligible token savings for the Qwen MoE model, which is not specialized for mathematical reasoning and tends to generate short, incorrect responses (Figure 1). For models with limited capability and short chains of thought, such as Qwen MoE, early-stopping provides little benefit.

	GSM8K	MATH-500	AIME
DSR	12.5/-0.5	9.4/-0.5	18.3/-1.5
DSL	0.9/+0.5	5.9/0	15.4/-0.5
Qwe	0.0/0	0.0/0	1.3/0

Table 2: Average token savings and accuracy degradation after applying *Guardian* on the full attention algorithm. Notation follows Table 1.

Cost of LZ77. We generate random strings of up to 128k tokens and measure the cost of LZ77 compression, which is approximately 34 ms. Because random strings exhibit minimal repetition and are more expensive to compress than natural language text, and because 128k tokens far exceed the sequence lengths used in our experiments, this measurement represents an upper bound. The resulting cost is comparable to the latency of decoding a single token (Zheng et al., 2024). With $f = 250$, compression is invoked once every 250 decoding steps, making the overall overhead negligible.

Token savings on correct and wrong cases. Table 3 reports token savings separately for correct and incorrect test cases, revealing two key observations. First, *Guardian* derives most of its effectiveness from terminating incorrect generations that would otherwise continue indefinitely; to avoid overstating this effect, we truncate such generations at twice the length of sequences produced under full attention (see the caption of Figure 1). Second, *Guardian* also reduces tokens in correct cases where the model has already produced the correct answer but continues to generate redundant reasoning, repeatedly rechecking the solution due to loss of earlier context.

Token savings on full attention. Figure 2 shows that even sequences generated with full attention achieve low compression ratios, indicating substantial redundancy. Table 2 further demonstrates that the early-stopping algorithm *Guardian* reduces token usage under full attention as well. This behavior is analogous to Chain-of-Thought

(CoT) compression approaches that aim to mitigate ineffective reasoning patterns arising from training data artifacts, human preferences for verbose explanations, or reward hacking (see Section 7). We therefore conclude that *Guardian* is applicable beyond sparse-attention settings and can be used more generally to address prolonged CoT generation. A direct comparison between *Guardian* and existing CoT compression approaches is left for future work.

7 Related Work

7.1 Sparse-Attention Algorithms

Sparse-attention algorithms exploit the fact that only a small subset of tokens receives significant attention scores, and we categorize them along two dimensions. (1) Inference stage. Some algorithms primarily accelerate the prefill stage, such as MInference (Jiang et al., 2024), FlexPrefill (Lai et al., 2025), and SpargeAttention (Zhang et al., 2025), while others focus on the decode stage, including H_2O (Zhang et al., 2023), StreamingLLM (Xiao et al., 2024b), Quest (Tang et al., 2024), InfLLM (Xiao et al., 2024a), DuoAttention (Xiao et al., 2025), and RaaS (Hu et al., 2024). (2) Training dependency. Training-aware algorithms incorporate sparsity into the model architecture and are trained jointly with the model, such as DeepSeek NSA (Yuan et al., 2025) and DSA (Liu et al., 2024). In contrast, training-free algorithms operate as plug-ins and can be readily applied to pretrained models; examples include H_2O, StreamingLLM, Quest, InfLLM, DuoAttention, and RaaS.

In this paper, we focus on Post-Training Sparse attention in the Decode stage (PTSD) for three reasons. First, the decode stage largely determines model performance, particularly in long-reasoning tasks, making it a critical target for optimization. Second, post-training sparsity is plug-and-play and can be integrated into existing models without retraining, making it practical for real-world deployment. Third, abundant sparse-attention algorithms fall into the PTSD category, making it a natural and impactful focus of study.

7.2 Chain-of-Thoughts Compression

Chain-of-thought (CoT) reasoning enhances model capability by enabling step-by-step inference that incrementally connects the prompt to the final answer. However, CoT can become unnecessarily long and redundant due to suboptimal training practices, such as reinforcement learning setups that inadvertently encourage verbose reasoning through reward hacking. Consequently, a line of research has emerged to focus on compressing prolonged CoT sequences.

Prior work on CoT compression, often referred to as long-to-short reasoning, can be broadly categorized into two classes. Training-aware approaches integrate compression into the training process, including ThinkPrune (Hou et al., 2025), DLER (Liu et al., 2025), and O1-Pruner (Luo et al., 2025). In contrast, post-training approaches operate on pretrained models, such as Answer5 (Liu and Wang, 2025), HALT-COT (Sun et al., 2025), and UnCertT (Zhu et al., 2025).

While this paper also targets CoT compression, the root causes of lengthy reasoning differ. First, in our setting, lengthy CoT arises from the use of sparse-attention algorithms, which induce repeated information loss and reconstruction. Second, by contrast, in long-to-short reasoning research, lengthy CoT typically stems from ill-reasoning patterns caused by data quality issues, human preference biases, or reward hacking.

These two root causes are largely orthogonal: when CoT is already lengthy due to ill patterns, sparse attention can further exacerbate its length. Although *Guardian* is designed to compress lengthy CoT induced by sparse attention, we observe that it also generalizes to ill-patterned CoT (Table 2), which we leave for future exploration.

8 Conclusion

Sparse-attention algorithms speed up each decode step but can hurt end-to-end efficiency because information loss during sparse decoding leads to repeated generation and longer outputs. We identify this problem as "Lil", quantify redundancy with compression ratio, and propose *Guardian* to stop decoding when information loss exceeds information gain. *Guardian* reduces unnecessary token generation without harming output quality. Beyond sparse decoding, *Guardian* can also be applied to general cases of prolonged Chain-of-Thought (CoT) generation. In future work, we plan to study *Guardian* on the prolonged CoT, which often arises from flawed reasoning patterns rather than the information loss and reconstruction characteristic of the Lil problem.

Limitations

Our work in this paper has the following major limitations.

Evaluation on a limited set of datasets and models. Our evaluation covers only three models and three datasets. As such, the results may not generalize beyond these specific configurations. Although models with longer context lengths (e.g., Qwen2.5-Max, DeepSeek-r1) and datasets such as GPQA Diamond and Codeforces exist, exhaustive evaluation across all combinations is computationally prohibitive (Hu et al., 2023). As reported in prior work (Zhong et al., 2024), decoding a single token can take approximately 30 ms; thus, processing 16k tokens on an A100-80GB GPU requires around 8 minutes. Running 200 test cases would take over a day on a single GPU, making large-scale evaluation infeasible with limited resources. Despite these constraints, we select datasets spanning three levels of difficulty and models covering diverse architectures (dense vs. MoE), training recipes, and parameter scales. We therefore believe that the Lil problem is not an artifact of specific configurations but is universal across datasets and models.

Evaluation on a limited set of sparse-attention algorithms. Our evaluation covers only four sparse-attention algorithms, and thus the results may not directly generalize beyond this set. Nevertheless, we contend that the Lil problem is inherent to sparse-attention algorithms, as all assume sparsity patterns—i.e., that a few tokens are important—and may lose information (Section 4). We select diverse algorithms, including the pioneering H_2O, the widely-used streamingLLM (recently applied in GPT-OSS (OpenAI, 2025)), the state-of-the-art Quest, and its counterpart infLLM. Other algorithms, such as clusterKV and PQCache, differ only in how K vectors are grouped and do not alter the use of sparsity. Hence, we believe the Lil problem is not specific to configurations but universal across sparse-attention algorithms.

References

AIME. AIME. https://huggingface.co/datasets/di-zhang-fdu/AIME_1983_2024.

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. 2024a. LongBench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the Sixty-Second Annual Meeting of the Association for Computational Linguistics*, pages 3119–3137.

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. 2024b. LongBench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the Sixty-Second Annual Meeting of the Association for Computational Linguistics*, pages 3119–3137.

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

Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. 2024. ArkVale: Efficient generative LLM inference with recallable key-value eviction. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 113134–113155.

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. 2021. Training verifiers to solve math word problems. *CoRR*.

Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. 2024. DeepSeekMoE: Towards ultimate expert specialization in mixture-of-experts language models. In *Proceedings of the Sixty-Second Annual Meeting of the Association for Computational Linguistics*, pages 1280–1297.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR*, abs/2501.12948.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*.

Bairu Hou, Yang Zhang, Jiabao Ji, Yujian Liu, Kaizhi Qian, Jacob Andreas, and Shiyu Chang. 2025. Thinkprune: Pruning long chain-of-thought of llms via reinforcement learning. *CoRR*, abs/2504.01296.

Junhao Hu, Wenrui Huang, Weidong Wang, Zhenwen Li, Tiancheng Hu, Zhixia Liu, Xusheng Chen, Tao Xie, and Yizhou Shan. 2024. RaaS: Reasoning-aware attention sparsity for efficient llm reasoning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, pages 2577–2590.

Junhao Hu, Chaozheng Wang, Hailiang Huang, Huang Luo, Yu Jin, Yuetang Deng, and Tao Xie. 2023. Predicting compilation resources for adaptive build in an industrial setting. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, pages 1808–1813.

Hugging Face. Hugging Face. https://huggingface.co.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention. In *Proceedings of the Thirty-Eighth Annual Conference on Neural Information Processing Systems*, pages 52481–52515.

Xunhao Lai, Jianqiao Lu, Yao Luo, Yiyuan Ma, and Xun Zhou. 2025. Flexprefill: A context-aware sparse attention mechanism for efficient long-sequence inference. In *Proceedings of the 13th International Conference on Learning Representations*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. DeepSeek-v3 technical report. *CoRR*.

Shih-Yang Liu, Xin Dong, Ximing Lu, Shizhe Diao, Mingjie Liu, Min-Hung Chen, Hongxu Yin, Yu-Chiang Frank Wang, Kwang-Ting Cheng, Yejin Choi, Jan Kautz, and Pavlo Molchanov. 2025. DLER: doing length penalty right - incentivizing more intelligence per token via reinforcement learning. *CoRR*, abs/2510.15110.

Xin Liu and Lu Wang. 2025. Answer convergence as a signal for early stopping in reasoning. *CoRR*, abs/2506.02536.

Haotian Luo, Li Shen, Haiying He, Yibo Wang, Shiwei Liu, Wei Li, Naiqiang Tan, Xiaochun Cao, and Dacheng Tao. 2025. O1-pruner: Length-harmonizing fine-tuning for o1-like reasoning pruning. *CoRR*, abs/2501.12570.

OpenAI. OpenAI o1. https://openai.com/o1/.

OpenAI. 2025. gpt-oss-120b & gpt-oss-20b model card. *CoRR*, abs/2508.10925.

Renliang Sun, Wei Cheng, Dawei Li, Haifeng Chen, and Wei Wang. 2025. Stop when enough: Adaptive early-stopping for chain-of-thought reasoning. *CoRR*, abs/2510.10103.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 2024. QUEST: query-aware sparsity for efficient long-context LLM inference. In *Proceedings of the 41st International Conference on Machine Learning*, pages 47901–47911.

Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023. How practitioners expect code completion? In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1294–1306.

Jun Wang, Meng Fang, Ziyu Wan, Muning Wen, Jiachen Zhu, Anjie Liu, Ziqin Gong, Yan Song, Lei Chen, Lionel M. Ni, Linyi Yang, Ying Wen, and Weinan Zhang. 2024. OpenR: An open source framework for advanced reasoning with large language models. *CoRR*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the Thirty-Sixth Annual Conference on Neural Information Processing Systems*, pages 24824–24837.

Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 640–654.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Yong Jiang, Pengjun Xie, Fei Huang, Minhao Cheng, Shuai Wang, Hong Cheng, and Jingren Zhou. 2025. Resum: Unlocking long-horizon search intelligence via context summarization. *CoRR*, abs/2509.13313.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. 2024a. InfLLM: Training-free long-context extrapolation for llms with an efficient context memory. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, pages 119638–119661.

Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. 2025. DuoAttention: Efficient long-context LLM inference with retrieval and streaming heads. In *Proceedings of the Thirteenth International Conference on Learning Representations*.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024b. Efficient streaming language models with attention sinks. In *Proceedings of the 12th International Conference on Learning Representations*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *The 11th International Conference on Learning Representations*.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. 2025. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, pages 23078–23097.

Jintao Zhang, Chendong Xiang, Haofeng Huang, Jia Wei, Haocheng Xi, Jun Zhu, and Jianfei Chen. 2025. Spargeattention: Accurate and training-free sparse attention accelerating any model inference. In *Proceedings of the 42nd International Conference on Machine Learning*.

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 *Proceedings of the 37th Annual Conference on Neural Information Processing Systems*, pages 34661–34710.

Yu Zhao, Huifeng Yin, Bo Zeng, Hao Wang, Tianqi Shi, Chenyang Lyu, Longyue Wang, Weihua Luo, and Kaifu Zhang. 2024. Marco-o1: Towards open reasoning models for open-ended solutions. *CoRR*.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. 2024. SGLang: Efficient execution of structured language model programs. In *Proceedings of the 38th Annual Conference on Neural Information Processing Systems*, pages 62557–62583.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation*, pages 193–210.

Yuqi Zhu, Ge Li, Xue Jiang, Jia Li, Hong Mei, Zhi Jin, and Yihong Dong. 2025. Uncertainty-guided chain-of-thought for code generation with llms. *CoRR*, abs/2503.15341.

Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343.

A LZ77 Compression Algorithm

We describe the LZ77 algorithm in detail. LZ77 operates using a sliding window buffer of fixed length n, divided into two parts:

- A *search buffer* of length $n - L_s$, which holds recently encoded data and acts as a dynamic dictionary.
- A *look-ahead buffer* of length L_s, which contains the subsequence awaiting encoding.

The encoding process proceeds iteratively. At each step, the algorithm finds the longest prefix of the look-ahead buffer that matches a substring within the search buffer. This match is encoded as a triple (p, l, c), where:

- p is the offset (distance) to the start of the match in the search buffer,
- l is the length of the matched substring,
- c is the first character that did not match (the literal following the match).

After encoding, both buffers are advanced by $l + 1$ characters, moving the processed symbols into the search buffer for potential future matches.

To concretely illustrate the encoding mechanism, Figure 4 provides a step-by-step visualization of LZ77 compressing the example sequence `0040040042304237`. The visualization uses distinct color-coded regions: the *search buffer* (green) holds the recently processed data available for matching; the *look-ahead buffer* (yellow) contains the pending sequence to be encoded; unprocessed input is shown in blue; and data that has shifted out of the search window is marked in red.

The algorithm iterates the same core loop. Below is a trace of its execution:

1. The first eight characters (`00400400`) are loaded into the look-ahead buffer, while the search buffer is filled with zeros. The algorithm finds the longest match for the look-ahead buffer's prefix within the search buffer. The prefix `00` matches at multiple positions; an offset of $p = 0$ and a length of $l = 2$ are selected. The first mismatching character is `4`, yielding the output triple $(0, 2, 4)$. The buffers then advance by $l + 1 = 3$ characters.

2. After the shift, the new content in the look-ahead buffer is `40040042`. The longest match found is `004004`, starting at offset $p = 5$ in the search buffer with length $l = 6$. The following character is `2`, producing the triple $(5, 6, 2)$. The buffers advance by 7 characters.

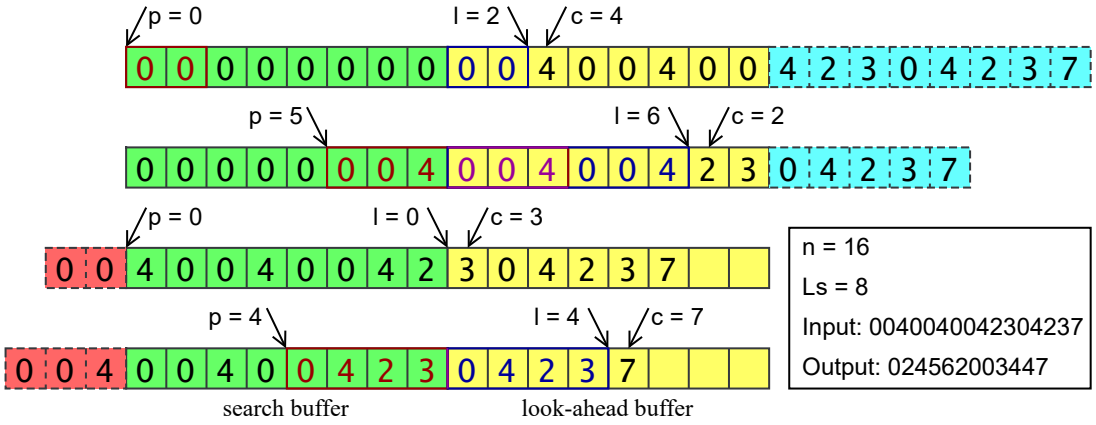

Figure 4: Illustration of the LZ77 algorithm.

		GSM8K				MATH-500				AIME			
	%	H2O	Sink	Quest	infLLM	H2O	Sink	Quest	infLLM	H2O	Sink	Quest	infLLM
DSR	128	83.0/67.7	59.3/68.3	0.9/7.7	3.8/6.8	77.4/82.0	79.7/84.3	7.3/11.7	3.0/8.8	89.1/93.7	84.8/92.6	37.2/18.2	17.6/14.1
	256	55.9/62.3	27.0/68.4	1.5/18.2	1.1/7.1	68.9/78.7	72.5/80.2	3.4/13.7	1.2/11.1	92.3/91.2	0.0/90.9	0.0/20.0	28.0/16.6
	512	23.7/57.9	20.1/64.9	0.3/25.7	0.0/13.2	52.0/67.5	32.0/73.6	2.1/18.4	3.1/14.3	78.1/86.5	90.6/87.8	1.2/24.5	5.2/25.0
	1024	15.1/61.9	6.8/63.9	0.0/50.4	0.5/41.4	26.6/64.8	16.7/65.7	0.9/23.6	1.0/26.2	49.8/79.9	30.7/80.6	0.6/25.0	5.0/36.9
DSL	128	21.6/22.8	3.8/8.6	0.0/2.6	0.0/0.5	83.9/75.2	41.5/48.3	19.8/14.3	1.9/4.0	91.9/91.0	59.2/71.0	30.7/23.1	0.0/4.9
	256	17.8/23.8	0.6/19.5	0.3/7.8	0.0/1.4	68.8/76.1	16.6/51.5	12.4/26.3	0.0/2.4	96.9/87.8	88.3/72.4	21.2/45.7	0.0/4.1
	512	4.9/29.5	1.1/16.4	0.7/6.0	0.0/0.9	46.5/74.2	7.7/49.7	10.4/35.0	0.1/6.8	73.5/87.6	22.0/69.3	33.8/55.8	6.7/10.4
	1024	3.7/20.5	0.8/10.0	0.4/4.0	0.0/1.0	21.0/71.7	8.2/52.0	7.4/42.5	1.1/12.2	60.3/84.2	23.4/69.3	21.6/54.2	2.3/20.7
Qwe	128	0.0/0.9	0.0/1.2	0.0/0.0	0.0/0.0	9.6/17.0	7.7/14.6	0.0/0.2	0.0/0.0	66.3/39.3	66.4/36.5	0.0/1.4	0.0/0.2
	256	0.0/0.8	0.0/2.5	0.0/0.0	0.0/0.0	7.0/21.2	2.0/15.4	0.0/0.0	0.0/0.5	48.6/42.6	0.0/31.7	0.0/1.2	0.0/0.8
	512	0.0/2.8	0.0/0.3	0.0/0.0	0.0/0.0	6.5/14.2	0.0/7.5	0.0/0.2	0.0/0.5	0.0/32.0	0.0/16.2	0.0/0.6	0.0/1.1
	1024	0.0/0.5	0.0/0.0	0.0/0.0	0.0/0.0	2.0/3.6	0.0/1.0	0.0/0.2	0.9/0.5	0.0/11.2	0.0/6.2	9.0/1.4	0.0/2.7

Table 3: Average token savings for correct and incorrect cases. Notation follows Table 1. Each cell reports (token savings on correct cases) / (token savings on incorrect cases).

3. The next character in the look-ahead buffer is `3`, which has no match in the current search buffer. This is encoded as a literal with $p = 0$, $l = 0$, and $c = 3$, resulting in $(0, 0, 3)$. The buffers advance by 1 character.

4. The final content in the look-ahead buffer is `04237`. The longest match is `0423`, found at offset $p = 4$ with length $l = 4$. The subsequent character is `7`, yielding the triple $(4, 4, 7)$. With the entire input processed, the algorithm terminates.

The complete encoded output is the concatenation of the triples from each iteration: `024562003447`.

A cornerstone of LZ77 is its theoretical connection between the compression ratio and the information-theoretic entropy of the source. For a stationary information source σ over a finite alphabet A, LZ77 (Ziv and Lempel, 1977) establishes the following bound when the buffer is sufficiently long:

$$h(L_s - 1) \leq \rho \leq h(L_s - 1) + \epsilon(L_s),$$

where ρ is the achieved compression ratio, $h(k)$ is defined as the per-symbol entropy $h(k) = \log_{|A|} |\sigma\{k\}|/k$ with $\sigma\{k\} = \{S \mid S \in \sigma \wedge |S| = k\}$, and $\epsilon(L_s)$ is asymptotically $\mathcal{O}(\log L_s/L_s)$. This classical result is fundamental to our analysis, therefore we restate its proof in Appendix B for completeness.

A simple rearrangement of this inequality yields:

$$\rho - \epsilon(L_s) \leq h(L_s - 1) \leq \rho.$$

This inequality confirms that the empirical compression ratio ρ provides a direct estimatebounded by a vanishing term $\epsilon(L_s)$of the true per-symbol entropy $h(L_s - 1)$ of the source. This provides a rigorous, information-theoretic foundation for using compression ratios as quantitative proxies for information content in our analysis.

<think>\nOkay, so I need to figure out which student has the greatest average speed based on the given Asymptote graph. ...*[about 1000 tokens]*... which is the slope of the line from (0,0) to their position.\n\nBut wait, perhaps each student's average speed is ...*[details omitted]*...n\nBut perhaps it's better to ...*[details omitted]*...\n\nBut since each student is at a single point, perhaps ...*[details omitted]*...\n\nTherefore, perhaps for each student, their average speed is their y-coordinate divided by their x-coordinate.\n\nTherefore, let's compute that.\n\nSo, for each student:\n\nEvelyn: y=4.5, x=1.25? Wait, no.\n\nWait, perhaps each student's average speed is their y-coordinate divided by their x-coordinate.\n\nBecause if you start from rest, then the average speed would be total distance divided by total time.\n\nTherefore, perhaps for each student, their average speed is y / x.\n\nTherefore, let's compute that.\n\nSo:\n\nEvelyn: y=4.5, x=1.25? Wait, perhaps not.\n\nWait, ...*[falls into repetition till the end]*...

(a) information loss > information gain

<think>\nOkay, so I need to figure out which student has the greatest average speed based on the given Asymptote graph. ...*[about 1000 tokens]*... which is the slope of the line from (0,0) to their position.\n\nBut wait, maybe not. ...*[about 500 tokens]*... Let me compute that: 4.5 divided by 1.25. Hmm, 1.25 is 5/4, so 4.5 divided by (5/4) is 4.5 * (4/5) = (4.5 * 4)/5 = 18/5 = 3.6. Wait, no, wait: 4.5 / 1.25 is equal to (4.5 * 100)/(1.25 * 100) = 450 / 1250. Wait, maybe better to compute it as:\n\n1.25 is 5/4, so 4.5 / (5/4) = 4.5 * (4/5) = (4.5 * 4)/5 = 18/5 = 3.6.\n\nWait, perhaps let's compute it as:\n\n1.25 is 1.25 hours? Wait, no, the units aren't specified, just the coordinates. So perhaps it's just a ratio.\n\nWait, perhaps it's better to ...*[details omitted]*...\n\nWait, perhaps better to ...*[details omitted]*... Wait, no.\n\nWait, perhaps I'm overcomplicating.\n\n...*[details omitted]*...\n\nBut ...*[about 1000 tokens]*... Thus, the student with the greatest average speed is \\boxed{Evelyn}.

(b) information loss < information gain

Figure 5: Examples of information loss and gain.

Applying the LZ77 algorithm, we compress all model outputs from our empirical study (Section 3). The average compression ratio computed across different stages of the generation process is presented in Figure 2. These results allow us to quantify how information content evolves—or stagnates—during prolonged decoding under sparse attention, directly testing our initial observation.

B Theorical Analysis for LZ77

To establish the relationship between the compression ratio and the entropy rate, we employ the following insight: since a limited information source only has constrained substrings, if the buffer length is sufficiently large to accommodate a substantial number of such substrings, then we can use the constraint to bound the compression ratio.

Consider a source σ defined over a finite alphabet A, constituted by a collection of strings with the property that certain specific substrings are prohibited. For each integer k, let $\sigma\{k\}$ denote the set of all length-k strings belonging to σ. The associated per-symbol entropy is given by $h(k) = \frac{1}{k}\log|\sigma\{k\}|$. Observe that it suffices to derive a bound for strings in $\sigma\{n - L_s\}$, as any such bound automatically extends to longer strings.

Take an arbitrary message $M \in \sigma\{n - L_s\}$. Suppose the algorithm partitions M into substrings $M = m_1 m_2 \dots m_N$, which are subsequently encoded into codewords $c_1, c_2, \dots, c_N$ of fixed length $\lceil \log_{|A|} L_s \rceil + \lceil \log_{|A|}(n - L_s) \rceil + 1 \triangleq L_c$. The compression ratio is therefore $\rho = \frac{L_c N}{n - L_s}$. Consequently, estimating ρ reduces to bounding N.

To estimate N, we analyze the partition in finer detail. For each p, let K_p be the number of substrings among $m_1, \dots, m_{N-1}$ having length p. Then

$$N = 1 + \sum_{m=1}^{L_s} K_m.$$

Thus the problem translates to bounding the counts K_m. Notice that if two substrings share the same length they must be distinct; i.e., $|m_i| = |m_j|$ implies $m_i \neq m_j$. So set $l = L_s - 1$ and define $\lambda = \lceil \log|\sigma\{l\}| \rceil$. We bound K_m in three regimes:

1. For $1 \leq m \leq \lambda$, the trivial bound $K_m \leq |A|^m \triangleq K'_m$ holds.

2. For $\lambda < m \leq l$, any substring of length m can be extended (in at least one way) to a string of length l in $\sigma\{l\}$. Hence $K_m \leq |\sigma\{l\}| \triangleq K'_m$.

3. For $m = l + 1$, we utilize the total length constraint

$$n - L_s = |m_N| + \sum_{m=1}^{l+1} m K_m,$$

which yields

$$K_{l+1} \leq \frac{1}{l+1}\Big(n - L_s - \sum_{m=1}^{l} mK_m\Big).$$

Substituting the upper bounds K'_m for K_m on the right-hand side produces an bound K'_{l+1} for K_{l+1}. Observing that the other terms K_m $(m \leq l)$ are individually overestimated, the bound on K_{l+1} obtained from the fixed total length constraint might be an underestimate. However, because each K_m contributes equally to N but the coefficient of K_{l+1} in the length sum is largest, the overall effect of substituting the overestimated K'_m into the bound for K_{l+1} still yields an overestimate for the total N.

Collecting these bounds, we obtain

$$N \leq K'_{l+1} + \sum_{m=1}^{l} K'_m \triangleq N'.$$

Now select n as follows:

$$\begin{aligned} n = &\sum_{m=1}^{\lambda} m|A|^m + \sum_{m=1}^{\lambda} m|\sigma\{m\}| \\ &+ (l+1)\Big(\sum_{m=1}^{\lambda}(l-m)|A|^m \\ &+ \sum_{m=1}^{\lambda}(l-m)|\sigma\{m\}| + 1\Big). \end{aligned}$$

With this choice we achieve $N' = \frac{n-L_s}{l}$, leading to the compression ratio bound

$$\rho \leq \frac{L_c}{l} = \frac{L_c}{L_s - 1}.$$

A trivial lower bound is $\rho \geq \frac{L_c}{L_s}$; hence the derived upper bound is reasonably tight.

Furthermore, note that the codeword length satisfies $L_c \leq 3 + \log(L_s - 1) + \log(n - L_s)$. From the definition of n,

$$\begin{aligned} n - L_s = l \cdot \Bigg[&\sum_{m=1}^{\lambda}(l-m)|A|^m \\ &+ \sum_{m=\lambda+1}^{l}(l-m)|\sigma\{l\}| \\ &+ \sum_{m=1}^{\lambda}|A|^m + \sum_{m=\lambda+1}^{l}|\sigma\{l\}|\Bigg]. \end{aligned}$$

Using the inequality $|A|^m \leq |\sigma\{l\}|$ for all $m \leq l$, we simplify to obtain

$$n - L_s \leq \frac{1}{2}\, l^2(l+1)\,|\sigma\{l\}|.$$

Substituting this into the bound for L_c gives

$$L_c \leq (L_s - 1)\Big[h(L_s - 1) + \epsilon(L_s)\Big],$$

where the error term is

$$\epsilon(L_s) = \frac{1}{L_s - 1}\Big(3 + 3\log(L_s - 1) + \log\frac{L_s}{2}\Big).$$

C Checklist-Related Issues

Three datasets, GSM8k (MIT), MATH500 (MIT), AIME (MIT), and three models, DeepScaleR 1.5B Preview (MIT), DeepSeek-R1-Distill-Llama-8B (MIT), Qwen1.5-MoE-A2.7B-Chat (tongyi-qianwen) are used with their intended usage scenarios. We retrieve all models and datasets from Hugging Face, where detailed documentation, including parameter sizes and model architectures, is provided. We manually checked the data and believe there is no personal information misused.

We used ChatGPT to check the grammar of the texts.

To the best of our knowledge, we believe our work does not pose risks that harm any subgroup of our society.