STS: Efficient Sparse Attention with Speculative Token Sparsity

Ceyu Xu*
The Hong Kong University of Science and Technology

Yongji Wu†
UC Berkeley

Jiangnan Yu*
The Hong Kong University of Science and Technology

Yuan Xie
The Hong Kong University of Science and Technology

Abstract

The quadratic complexity of attention imposes severe memory and computational bottlenecks on Large Language Model (LLM) inference. This challenge is particularly acute for emerging agentic applications that require processing multi-million token sequences. We propose STS, a sparse attention mechanism that requires no model retraining. STS leverages the key insight that **tokens identified as important by a smaller draft model are highly predictive of important tokens for a larger target model.** By integrating into speculative decoding frameworks, STS repurposes the draft model's attention scores to dynamically construct a token-and-head-wise sparsity mask. This mask effectively prunes the expensive attention computation in the target LLM. Our evaluation shows that STS achieves a 2.67× speedup operating at approximately 90% sparsity on representative benchmark NarrativeQA, maintaining negligible accuracy degradation compared to dense attention. STS establishes a new state-of-the-art on the sparsity-accuracy trade-off, outperforming prior techniques by enabling higher sparsity levels for a given accuracy budget.

1 Introduction

The power of the attention mechanism [27] originates from its ability to construct a globally aware context by computing relevance scores between every pair of tokens in the sequence. This "attend-to-everything" approach is a double-edged sword, enabling unprecedented model capability while incurring quadratic complexity as the sequence scales: On the one hand, this architecture has become ubiquitous in modern AI, serving as the foundation for the emergence of advanced machine intelligence. Conversely, the necessity of attending to all preceding tokens imposes severe computational and memory-access overheads, which become prohibitive in applications that require long sequence lengths.

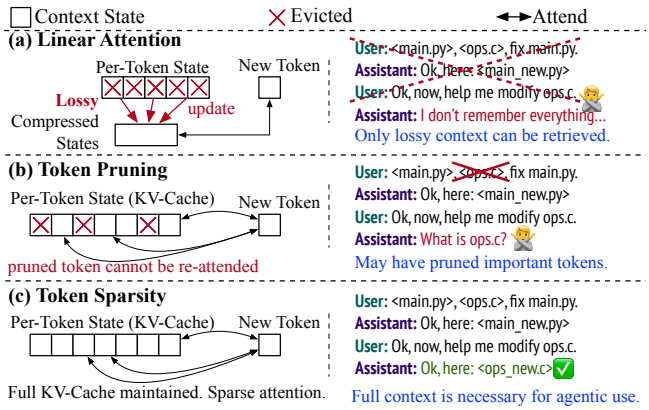


Figure 1: A taxonomy of methods for reducing the computation cost of dense attention: (a) Linear Attention, (b) Token Pruning, and (c) Token Sparsity.

To mitigate these costs, prior work [5, 9, 10, 14, 26, 32, 34, 37, 39] focuses on designing an approximation of the attention mechanism that trades a bit of accuracy for lower computational cost. One class of approximation is linear attention [5,10,14,28] or token pruning [32,39] as shown in Figure 1 (a) and (b). However, these methods often cause irreversible information loss, which is risky because the importance of a token is dynamic, so a token that is irrelevant at one step may become critical later. Consequently, lossy methods can cause model accuracy degradation when the model requires historical information that was previously discarded. A more robust alternative is *Token Sparsity* [26, 34]. This approach maintains the full KV cache but dynamically selects a small subset of critical tokens for the current query. The central question thus becomes how to design an algorithm that accurately and efficiently identifies this critical subset of tokens for the current query, without prematurely discarding the full context needed for future queries.

Accurately identifying the importance of a specific token without performing the full attention computation remains a non-trivial challenge. Existing approaches typically rely on

*Both authors contributed equally.
†Corresponding author.

heuristics [26, 34] and suffer from three fundamental limitations that impede practical deployment. First, lightweight selection algorithms often achieve suboptimal accuracy; when critical tokens are missed, they must retain significantly more tokens than necessary to preserve model quality, or else accuracy degrades. Second, many methods reduce the search space by partitioning tokens into blocks and selecting entire blocks rather than individual tokens, which limits the maximum achievable sparsity. Third, the token selection process incurs computational overhead that increases LLM inference latency, often offsetting the performance gains from reduced attention complexity.

In this paper, we address these challenges with Speculative Token Sparsity (STS), a sparse attention algorithm that is accurate, efficient, and supports per-token granularity by leveraging the speculative decoding framework. The efficacy of STS stems from a key observation: for a given input, the attention patterns of two different models exhibit strong correlation, especially when the models belong to the same family. Leveraging this insight, STS repurposes the small draft model so that, in addition to its standard role of proposing new candidate tokens, it now also generates a dynamic sparsity mask from its own attention computations. The large target model then uses this mask to sparsify its attention computation. Consequently, the prohibitively expensive dense attention is confined to the small draft model. In contrast, the target model performs a highly efficient sparse attention operation, drastically reducing both computational and memory bandwidth requirements with minimal impact on model accuracy.

Deriving the sparsity mask from the draft model offers several advantages. First, STS is training-free, simplifying its deployment. Second, its seamless integration with speculative decoding frameworks makes it well-suited for low-latency inference. STS benefits from reduced attention computation due to sparsification, and unlike prior approaches that perform token selection immediately before each attention computation, STS executes token selection on the smaller draft model asynchronously with respect to the target model's execution path. Because this selection finishes before the target model needs the sparsity masks, its latency can be hidden within the main execution pipeline. In addition, this "known-in-advance" property distinguishes STS from just-in-time mask generation methods, which enables further optimizations, such as proactive data prefetching to hide communication latency in a KV-cache offload scenario. Moreover, while existing methods often revert to dense computation during the prefill phase, negating performance gains in multi-turn agentic interactions [20, 29], STS maintains sparsity across both prefill and decode phases. Finally, compared to existing token sparsity schemes, STS achieves a state-of-the-art accuracy-sparsity trade-off, demonstrating that the draft model serves as a highly effective proxy for token selection.

This paper makes the following contributions:

1. We propose STS, a training-free sparse attention mechanism applicable to both prefill and decode stages. It significantly reduces the memory bandwidth and computational requirements of LLMs with negligible impact on model accuracy.

2. We implement STS in vLLM [16], demonstrating its seamless integration into speculative decoding frameworks. Our evaluation shows that STS achieves a 2.67× speedup operating at approximately 90% sparsity on representative benchmark NarrativeQA, maintaining negligible accuracy degradation compared to dense attention.

3. We evaluate STS against state-of-the-art token sparsity techniques and show that it achieves a superior accuracy-sparsity trade-off, enabling higher sparsity for a given accuracy budget (and vice versa).

4. We identify and analyze the "known-in-advance" property of STS, where the sparsity mask is determined before the target model's computation. We show that this unique feature enables further latency optimizations and opens new avenues for future, more clever memory orchestrations.

2 Background

This section first provides context on speculative decoding. We then discuss the taxonomy of attention approximation techniques, comparing their respective advantages and disadvantages to contextualize our proposed approach.

2.1 Speculative Decoding

The autoregressive nature of large language models (LLMs) makes generation a sequential process, in which each model generates only one new token at a time. For each new token, the model must scan its entire set of parameters and the complete Key-Value (KV) cache of all preceding tokens. This makes the process extremely memory-intensive, causing overall decoding latency to be largely bottlenecked by GPU memory bandwidth.

Speculative Decoding [3, 17] aims to solve this issue by breaking the inherent sequentiality, enabling the LLM to decode multiple tokens in a single forward pass. It achieves this by using a smaller, faster "draft" model to generate a sequence of new candidate tokens, while the larger "target" model then behaves like a verifier, checking all candidates in a single parallel decoding step. The reason this is effective is that although the draft models are smaller and less capable, they can still generate a relatively correct sequence, especially when some tokens are not that hard to predict. In a successful speculation, a prefix of candidate tokens is accepted, allowing the model to take multiple steps in a single forward pass and achieving a significant speedup. When a prediction fails,

forward progress is still guaranteed. The system accepts correctly guessed tokens up to the point of the mismatch, then appends the single correct token produced by the target model for that position. In reality, the overhead of the draft model is so small that it becomes almost free, resulting in pure latency reduction.

The core idea of speculative decoding, therefore, is to use a small model to guide a large model. In our work, STS, we learn from this principle and extend it, positing that the small model's guidance can be applied not only to predicting output tokens but also to approximating the expensive internal computations of the large model.

Table 1: A comparison of methods to reduce the computational cost of dense attention. (St. LLM stands for StreamingLLM [32] and TidalDec. stands for TidalDecode [34] and Lin. Attn. stands for Linear Attention [28])

Method	Lin. Att.	St. LLM	H2O [39]	Quest [26]	Tid. Dec.	NSA [37]	STS*
Lossless Context	N	N	N	Y	Y	Y	Y
Sparse Prefill	-	Y	N	N	N	Y	Y
Sparse Decode	-	Y	Y	Y	Y/N	Y	Y
No Extra Latency	-	Y	Y	N	N	N	Y

2.2 Linear Attention and Token Pruning

While techniques like speculative decoding can mitigate the latency of auto-regressive generation, the underlying $O(N^2)$ complexity of the attention mechanism remains the primary bottleneck for scaling LLMs to long context windows. To circumvent this quadratic complexity, two prominent strategies that perform lossy context state compression have emerged: linear attention and token pruning. Linear attention methods, such as Linformer [28], Performer [5], Reformer [14], and Mamba [10] approximate the attention softmax operation with transformations that have linear time and space complexity. Token pruning, employed by methods like H2O [39] and **Scissorhands** [21], is a more direct approach that compresses or permanently discards tokens from the key-value (KV) cache based on specific heuristics, while keeping the core attention computation unmodified.

Both strategies, however, share a fundamental limitation: they perform irreversible, lossy compression of the context. As articulated in multiple related works [26, 32, 37, 39], the importance of a token is dynamic, and it is impossible to know what information will be critical for future queries. This limitation is particularly acute in multi-round agentic workloads, as illustrated in Figure 1(a) and (b). In these scenarios, the model must decide what context to discard or compress in one turn, without knowing what information will be required to respond to environmental feedback in a subsequent turn. Prematurely removing a token that later becomes crucial can lead to a catastrophic loss of model capability, especially given the "Lost in the Middle" phenomenon [19] where critical information often resides in the middle of the context.

Nevertheless, the appeal of these methods lies in their potential to optimize not only the attention computation but also the size of the KV cache itself—a key challenge in long-context serving [16]. However, because maintaining a lossless context is essential for robust performance, the risk of severe accuracy degradation makes these approaches ill-suited for complex, unpredictable applications. This motivates the exploration of token sparsity, a lossless alternative.

2.3 Token Sparsity

Token sparsity presents a more promising alternative that preserves the full, lossless KV cache. Instead of compressing the context, these methods exploit the natural sparsity of the attention matrix [2, 4, 38] by computing attention on a dynamically selected, small subset of important tokens for each query. This approach maintains the full context for future steps while reducing the computational cost of the current step. The central challenge, however, lies in efficiently and accurately identifying this critical subset of tokens.

Current methods for token sparsity mainly differ in their strategies for token selection, often depending on a "proxy" representation to assess the significance of tokens. We provide a graphical illustration of several approaches in Figure 2 and a qualitative comparison of their trade-offs in Table 1.

Heuristic-Based Selection. Some methods identify important tokens using runtime heuristics [22, 26]. Quest, depicted in Figure 2(a), groups tokens into blocks and computes a compact "summary vector" for each block through channel-wise Min/Max. During generation, these summary vectors act as a proxy to efficiently identify the most relevant blocks of the KV cache. Similarly, TidalDecode [34], shown in Figure 2(c), posits that attention patterns from early model layers can predict token importance for later layers. It computes dense attention for the first few layers and leverages the resulting scores as a heuristic to sparsify computation in subsequent layers.

Trained Selection. An alternative approach involves training a dedicated gating network to select important tokens, rather than relying on hand-crafted heuristics. For instance, DeepSeek's Native Sparse Attention (NSA) [37], shown in Figure 2(b), uses a trained gating network to generate sparsity masks. While this can yield highly accurate masks, it requires extensive model retraining. This makes the approach impractical for deploying on existing, pre-trained dense models and infeasible for developers without access to large-scale computational resources.

Performance Considerations. A significant drawback of these approaches is the selection overhead they introduce.

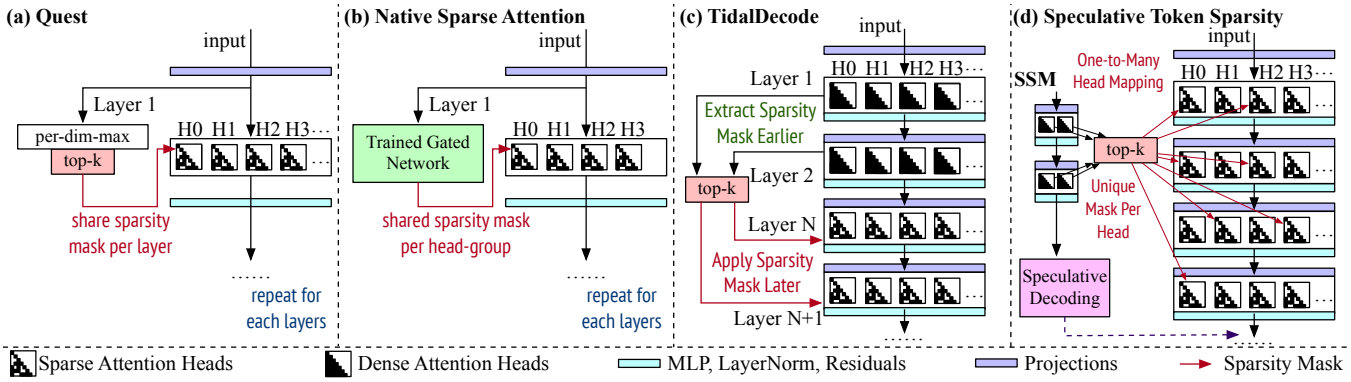


Figure 2: STS vs. Other Token Sparsity Works. The key difference is what "proxy" is used for selecting the important tokens.

The heuristic computations or gating network inferences are performed just-in-time, adding latency that can offset the performance gains from sparsity. Furthermore, a critical limitation, summarized in Table 1, is that many existing methods like Quest and TidalDecode support sparsity only during the decode phase, requiring a dense prefill. This design is fundamentally incompatible with multi-turn agentic workloads for two reasons.

First, agentic workloads are often prefill-bound; the input context from the environment can be long, while the generated output (e.g., a tool call) may be short. A decode-only optimization thus fails to accelerate the most expensive phase. Second, the multi-turn nature of agentic interactions involves a continuous cycle of prefill (processing new information) and decode (generating a response). A decode-only sparse method creates a state mismatch: the sparsely computed KV cache from one turn's decode step cannot be used by the dense prefill required in the next turn. This incompatibility effectively breaks inter-turn KV caching, forcing costly recomputation and negating a primary benefit of context management in multi-turn sessions. Our work, STS , is designed to overcome these limitations.

3 Motivation

In recent years, agentic workflows have emerged as a central serving use case, consuming a vast majority of inference tokens due to their iterative, autonomous nature [20,35]. However, serving these workloads efficiently poses distinct challenges that existing methods fail to address. While STS introduces unique architectural features that benefit any standard LLM infrastructure, its design is particularly advantageous for serving as a model for *agentic applications*.

The Challenges of Agentic Serving: Unlike simple chat applications, agentic workloads typically involve longer sequence lengths and often exhibit more complicated request patterns. An agent does not merely generate text; it engages in a continuous loop of reasoning and interacting with the environment. This results in a cycle involving long initial prefills, "prefill-after-decode" (processing environmental feedback), and multi-turn conversations that require KV caches to persist for extended periods.

Furthermore, these systems are highly *latency-critical*. An agent must wait for the LLM's decoding to finish before it can further interact with the environment. Consequently, the serving latency directly dictates the timeliness of the agent's actions and the overall task-level timeliness.

Finally, the need for long sequence lengths in agentic applications requires massive amounts of memory for KV-cache storage. In practice, such constraints make *KV-cache offloading* a necessity as agents often need to wait for environment responses (e.g., shell commands) [25], allowing the system to serve other users in the interim. However, because it is impossible to keep all active KV caches in GPU memory, the system must frequently swap contexts between CPU and GPU, making the latency caused by swapping a significant concern.

The Failure of Existing Methods: Current optimization techniques fail to meet these demands in terms of accuracy, capabilities, and performance.

First, methods relying on *lossy context* (e.g., token pruning or compression) are effectively unusable for agents. In an autonomous workflow, it is impossible to predict which specific piece of historical information the agent will need in a future step. Proactively pruning context is dangerous, as it often leads to irreversible information loss that damages task-level accuracy and reasoning capabilities.

Second, existing lossless sparse attention methods typically support sparsity only during the decoding phase, with their *prefill stages remain 100% dense*. While this may suffice for single-turn chat, it is catastrophic for multi-turn agentic loops. In a multi-turn environment, the KV cache generated by a [Dense Prefill] $\rightarrow$ [Sparse Decode] sequence in turn N must be reused in turn $N+1$. However, because these frameworks do not support sparse prefill, the system cannot execute a sequence of [Sparse Decode] $\rightarrow$ [Sparse Prefill] $\rightarrow$ [Sparse Decode]. Instead, the context generated sparsely in

previous turns must be re-processed densely when new prefill requests arrive (e.g., appending tool outputs). This constant re-computation offsets the gains from sparse decoding, failing to alleviate the raw compute and memory bottlenecks.

The STS Solution: STS is designed to bridge this gap by decoupling mask generation from the target model's execution. By utilizing a draft model to generate sparsity masks asynchronously, STS eliminates latency overhead on the critical path of the target model inference, thereby meeting the strict demands of agentic serving.

Crucially, STS provides the *"known-in-advance"* property. Unlike methods that compute the sparsity masks just-in-time, STS determines the sparsity pattern before the target model begins computation. This provides a unique opportunity to **prefetch offloaded KV-caches**. Moreover, STS is training-free and targets practical deployment, naturally integrating into existing vLLM infrastructures and speculative decoding frameworks to provide a robust solution for the next generation of AI agents. Finally, as we will show in the following sections, STS achieves the state-of-the-art sparsity, model-level accuracy, and performance at the same time.

4 STS Design

This section presents the algorithmic details of STS. The overall workflow, illustrated in Figure 3, consists of a one-time offline head mapping phase followed by an online inference phase. We organize our discussion around these stages.

The first stage involves generating a static mapping between the attention heads of the smaller draft model and those of the larger target model. This process itself has two steps. We first collect a set of attention weight traces from both models on a representative dataset, as shown in Figure 3(a). We then use these traces to compute a correlation-based mapping, pairing each target model head with a draft model head as depicted in Figure 3(b). The algorithms for trace collection and head mapping are detailed in Section 4.1.

Once the mapping is generated, STS performs sparse inference. We present the inference flow in two parts for clarity. First, in Section 4.2, we describe the logical flow of computation, explaining how the draft model's attention scores are used to create sparsity masks for the target model (Figure 3(c)). Second, in Section 4.3, we explain how STS integrates into a speculative decoding system, detailing the necessary mechanisms to handle the asynchronous execution of the draft and target models (Figure 3(d)).

4.1 Attention Head Mapping

A central challenge in using a small model to guide a large one, an approach inspired by speculative decoding, is the mismatch between the two. Differences in hidden dimensions and the number of layers make it challenging to establish a direct correspondence between their internal representations. The STS

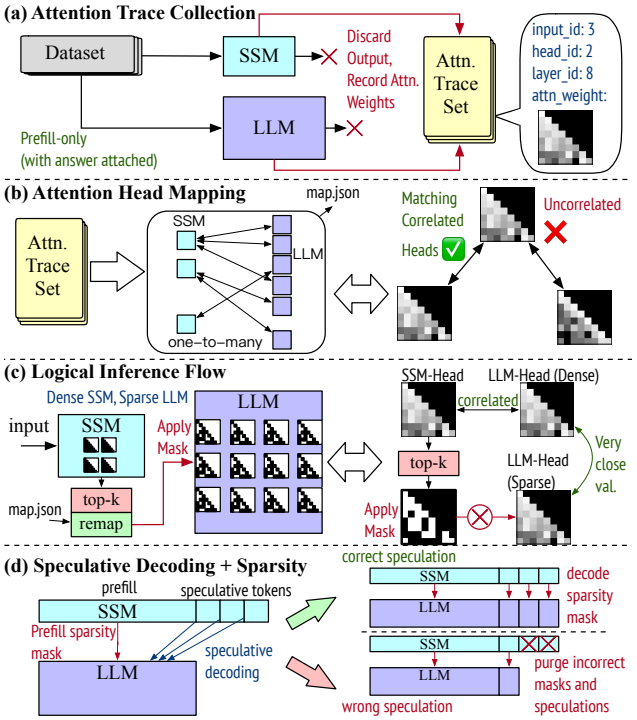


Figure 3: Overview of the STS Workflow.

algorithm overcomes this by leveraging a key similarity of the Transformer architecture: the attention mechanism. While models vary in size, the attention weight matrix for any head, regardless ofwhether it comes from the small draft model or the large target model, is consistently shaped $N \times N$, where N is the sequence length. A model's size only determines the number of attention heads, with larger models having more than smaller ones, but does not change the $N \times N$ nature of the attention. This insight simplifies the problem from one of incompatible shapes to one of mapping a smaller set of draft model heads to a larger set of target model heads. STS can therefore establish this mapping based on the correlation between the heads' attention patterns.

The first step in creating this mapping is an offline trace generation phase, as illustrated in Figure 3(a). The trace generation phase requires a small *trace generation dataset* to be prepared. During trace generation, we perform a prefill-only forward pass for each sample through both the draft and target models. For each forward pass, we record the attention weight matrices from each head in both models and store them in a trace file for the subsequent mapping step. To demonstrate the generalizability of our approach, we use only the first 100 samples from the WikiText-2 dataset [23] as our *trace generation dataset* for all evaluations in this paper.

Once the attention traces are collected, the next step is to construct a mapping from the heads of the target model to those of the draft model. Because the target model has more attention heads than the draft model, this process results

Algorithm 1 STS Head Mapping Construction

procedure FINDHEADMAPPING(A_D, A_T, k)
  ▷ A_D, A_T: Attention traces from the draft and target.
  ▷ k: The number of top attention scores to select.
  $H_D \leftarrow \text{Heads}(A_D)$ ▷ Set of heads in the draft model
  $H_T \leftarrow \text{Heads}(A_T)$ ▷ Set of heads in the target model
  $\mathcal{M} \leftarrow \emptyset$ ▷ Initialize an empty map
  for each target head $h_t \in H_T$ **do**
    $max\_score \leftarrow -1$
    $best\_match \leftarrow \text{NIL}$
    $M_t \leftarrow \text{TopKIndices}(A_T(h_t), k)$
    for each draft head $h_d \in H_D$ **do**
      $M_d \leftarrow \text{TopKIndices}(A_D(h_d), k)$
      $score \leftarrow |M_t \cap M_d|$ ▷ Compute the total number of matching indices of the two heads
      if $score > max\_score$ **then**
        $max\_score \leftarrow score$
        $best\_match \leftarrow h_d$
      end if
    end for
    $\mathcal{M}[h_t] \leftarrow best\_match$
  end for
  return $\mathcal{M}$
end procedure

in a one-to-many mapping, where a single draft head may be responsible for generating the sparsity mask for multiple target heads. The objective is to pair heads that exhibit highly correlated attention patterns, as shown in Figure 3(b). While a standard metric like the Pearson correlation coefficient could measure the similarity of the traces, it aligns sub-optimally to our optimization target: STS 's effectiveness hinges on how well the draft model's 'top-k' attention scores predict the target's 'top-k' scores. Therefore, the ideal correlation metric must directly evaluate the similarity of the resulting token masks, not the raw attention values. Based on this reasoning, we developed the procedure detailed in Algorithm 1.

Our algorithm iterates through each attention head in the target model and finds the best-matching head from the draft model. The quality of a match is determined by the similarity of the sparsity masks they generate across the collected traces. Specifically, for a given target head, the algorithm identifies the draft head whose 'top-k' indices produce the largest intersection with the oracle 'top-k' indices from the target head. It is important to note that this process yields a unique mapping for each value of the sparsity parameter k. Since the mapping algorithm is run offline on a small trace set, it is computationally cheap and introduces no online overhead.

4.2 Inference Flow of STS

At runtime, STS leverages the draft model to guide the target model's sparse attention computation based on the head mappings already prepared. This process is illustrated in Figure 3(c). The inference flow begins with a standard, dense forward pass of the input sequence through the draft model. From this pass, the attention weights of each head are extracted. A 'top-k' operation is then applied to these weights to identify the indices of the most important tokens, creating a sparsity mask for each draft head. Subsequently, these masks are remapped using the offline-generated mapping dictionary, which assigns the correct sparsity indices to each corresponding head in the target model. Finally, the target model performs its forward pass, with its attention layers computing sparsely by attending only to the key-value pairs specified by the provided indices. The sparsity indices are generated at the token level, enabling fine-grained token-wise sparse attention. By applying top-k selection over blocks instead of individual tokens, STS can also support block-wise sparsity, allowing it to adapt to diverse deployment targets.

A key feature of STS is its applicability to both the prefill and the decode stages. The core logic remains identical, but the structure of the attention mask differs based on the computation being performed. During prefill, multiple query tokens attend to the input sequence, resulting in a two-dimensional attention pattern for each head. During the decode stage, a single new token attends to the entire key-value cache. Consequently, the draft model generates a one-dimensional sparsity mask for each head, indicating the most relevant tokens in the context for that single new query.

4.3 STS Integration into Speculative Decoding

Speculative decoding executes two models in tandem: a lightweight draft model proposes a bundle of candidate tokens while the heavyweight target model verifies them in one parallel decode step. STS piggybacks on this infrastructure by treating the draft model as an oracle for guiding the target model's attention sparsity. In each speculative round, the draft model runs over the speculated context and emits the attention logits that the STS controller converts into sparsity masks. When the target model begins its verification pass, it uses that mask to perform sparse attention. After the target finishes its verification pass, speculative decoding either accepts a prefix of the candidates or rejects them all. STS handles the two cases as follows:

- **Successful speculation:** When the target model accepts a chunk of speculatively decoded tokens, the corresponding sparse metadata has already been consumed during verification. No further action is required with the KV-cache for the accepted tokens continue to persist in the memory, and the runtime simply advances to the next round.
- **Speculation failure:** If the target rejects the proposal (for example, due to a mismatch on the first token), the runtime discards both the speculative tokens and the sparsity mask that was paired with them. The next speculative attempt

triggers a fresh draft forward pass, generating a new mask that reflects the updated context.

This coupling keeps STS strictly aligned with speculative decoding: masks are generated exactly once per speculative batch, consumed immediately by the target verification pass, and either retained implicitly through accepted KV cache states or thrown away with the rejected bundle. The asynchronous execution of the draft and target runners is bridged by the runtime described in Section 5, allowing sparse attention to remain functional even when the speculative verifier backtracks.

5 STS Implementation

We implement STS on top of vLLM [15] (v0.9.2) and FlashInfer [36] (v0.1.6). Our implementation comprises approximately 7,000 lines of Python and 2,400 lines of CUDA/C++.

Attention Kernels We extend FlashInfer's attention kernels to support arbitrary sparsity patterns at block granularity. Each attention head can be assigned with a different set of KV blocks to attend to, and is allocated to a thread block for computation. Split-K trick is applied for longer sequences to improve SM utilization [36]. The KV blocks scattered across the GPU global memory are first read into contiguous shared memory tiles. Attention computation is performed over the shared memory and is pipelined with shared memory loading. As STS computes the LLM's attention masks using the attention scores of the LLM, we also modify the full attention kernel to output the product between queries and keys into a pre-allocated buffer in global memory.

Drafting-Speculation Pipelining To minimize the overhead of generating the sparsity mask, we implement a pipeline execution mechanism where the forward of layer i of the SSM is overlapped with the computation of the sparsity masks of the LLM's attention heads that are mapped to that from layer $i-1$ from the SSM. The forward computation and mask generation are placed on separate CUDA streams, while their data dependency is resolved with CUDA event synchronization. The mask computation fully resides on GPU and is compatible with CUDA graphs for efficient execution.

6 Evaluation

In this section, we evaluate STS on its performance and accuracy. Our evaluation addresses three critical questions:

- **Performance (Section 6.2):** How does the theoretical sparsity degree of STS effectively translate into wall-clock speedups. How does the "know-in-advance" feature of STS help hide the latency during KV-cache offloading?
- **Fidelity & Robustness (Section 6.3):** Does STS preserve generation quality across diverse workloads, ranging from "needle-in-a-haystack" retrieval to complex reasoning?
- **Insights from Micro-Analysis (Section 6.4):** How does the draft-guided sparsity pattern align with the actual attention mechanisms of Large Language Models?

6.1 Experimental Setup

Hardware Testbed. All experiments are conducted on a high-performance server equipped with NVIDIA H20 GPUs (96GB HBM3 memory). To rigorously evaluate our offloading mechanism, the GPU is connected to the host CPU via a PCIe Gen5x16 interface, providing a theoretical bi-directional bandwidth of 128 GB/s. This setup represents a typical production environment for serving large-scale models.

Software & Baselines. We compare against a comprehensive suite of baselines:

- **Dense (Full Attention):** The gold standard for accuracy, but computationally expensive ($O(N^2)$).
- **Quest [26]:** Leverages query-aware sparsity for efficient long-context inference. It identifies critical tokens by estimating the importance of KV cache pages relative to the current query.
- **TidalDecode [34]:** A method that selects tokens based on spatial locality and sparse metrics.
- **StreamingLLM [32]:** A static sparsity baseline that keeps only the initial (sink) and recent tokens.

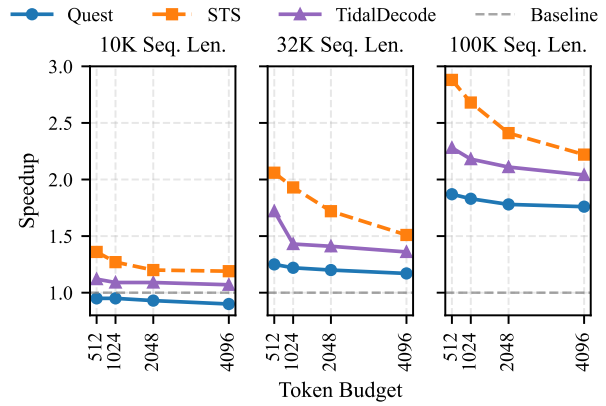
We use **Llama-3.1-8B-Instruct** as the target model and **Llama-3.2-1B-Instruct** as the draft model. Unless stated otherwise, experiments use greedy decoding (temperature=0) to ensure deterministic reproducibility.

6.2 End-to-End System Performance

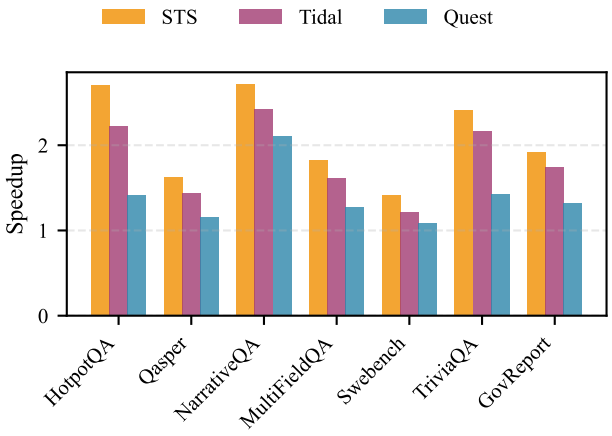
6.2.1 End-to-End System Speedup Analysis

Figure 4a and Figure 4b present the speedup of STS relative to the Dense baseline.

Scaling with Context Length. As illustrated in Figure 4a, STS demonstrates superior scalability as the sequence length grows from 10K to 100K. Even at a relatively short context of 10K, STS already yields a 1.36× speedup over the dense baseline, outperforming other sparse attention baselines. This advantage becomes more pronounced at 32K context, where the speedup rises to 2.06×. The gap widens significantly in the extreme setting of 100K context length; with a token budget of 512, STS achieves a remarkable **2.88× speedup**. In contrast, Quest and TidalDecode plateau at lower speedups



(a) Speedup across varying sequence lengths



(b) Speedup on real-world workloads

Figure 4: **[System Performance]: End-to-end speedup analysis.** (a) STS scales efficiently up to 100K context length. (b) STS maintains performance advantages on realistic benchmarks like LongBench and SwBench.

as the sequence length increases. This widening gap stems from the overhead of "Just-in-Time" selection. Methods like Quest require loading coarse-grained Q and K metadata from HBM to compute importance scores *during* the target model's execution. As context length grows (e.g., from 10K to 100K), this memory bandwidth consumption becomes a bottleneck. STS , however, decouples mask generation: the sparsity mask is pre-computed by the draft model. This allows the target model to execute sparse attention kernels immediately without stalling for metadata analysis, ensuring consistent speedup gains across all context lengths.

Real-world Workloads. We further evaluated STS on realistic datasets, including LongBench and SweBench [13], as shown in Figure 4b. For these experiments, we standardized the active token budget to **4096**. This choice is empirically grounded; our subsequent accuracy benchmarks (see Section 6.3) confirm that a 4096 budget is sufficient to preserve generation quality for these workloads.

Under this setting, STS consistently delivers the highest speedup. We highlight our evaluation methodology on **SweBench**, which involves complex code generation tasks. Instead of using synthetic inputs, we adopted a trace-driven approach: we sampled 100 instances from the dataset, collected their execution traces to determine the average sequence length and output token counts, and measured the speedup based on these realistic profiles. Results indicate that STS outperforms TidalDecode by a significant margin on SweBench. This suggests that STS 's draft-based selection is far more robust to the irregular and long-range attention patterns typical of code repositories compared to locality-biased heuristics.

6.2.2 Hiding Latency via Prefetching

A critical bottleneck in long-context serving is GPU memory capacity. When the KV cache exceeds HBM limits, systems must offload data to CPU RAM. Standard offloading incurs severe latency penalties due to the PCIe bandwidth bottleneck. With the "know in advance" feature of STS , part of the latency from offloading can be hidden through prefetching. Since the draft model runs ahead of the target model, we know exactly which KV blocks will be needed *before* the target model begins its layer computations. We leverage this lookahead to issue asynchronous prefetch requests over the PCIe Gen5 bus, transferring blocks from CPU memory to GPU memory before they are strictly needed. In memory-constrained scenarios where the full KV cache cannot fit in GPU memory, this prefetching capability significantly reduces the performance overhead of CPU offloading.

We evaluate the effectiveness of prefetch-enabled KV-cache offloading by comparing three strategies:

- **Full GDDR:** The entire KV-cache resides in GPU memory, serving as the ideal performance baseline.
- **On-Demand:** KV-cache blocks are stored in CPU memory and transferred to the GPU synchronously only when needed, blocking computation.
- **Prefetch (STS):** STS predicts the required KV-cache blocks several tokens ahead and initiates asynchronous transfers that overlap with computation.

Figure 5 presents the latency comparison across varying context lengths (1K–512K tokens) and sparsity levels (1/8, 1/16, and 1/32). The **On-Demand** approach incurs substantial overhead, with latency increasing by 14–19× compared to the GPU-only baseline due to synchronous memory transfers. In contrast, the **Prefetch** approach reduces this overhead to 6–8× by effectively overlapping data transfers with kernel execution. This represents a roughly **3× reduction** in effective latency. These results demonstrate that STS 's ability to predict future sparse attention patterns enables effective hiding

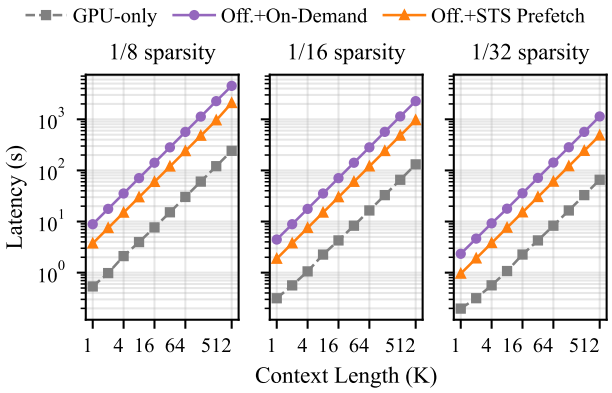


Figure 5: **[System Optimization]**: Latency of KV-cache offloading strategies.

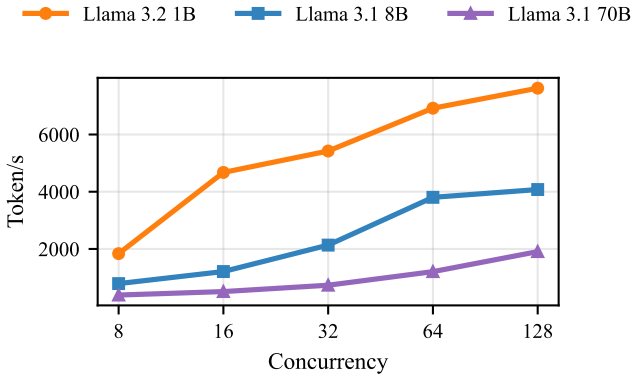


Figure 6: **[Draft Overhead Analysis]**: Throughput scalability under concurrency.

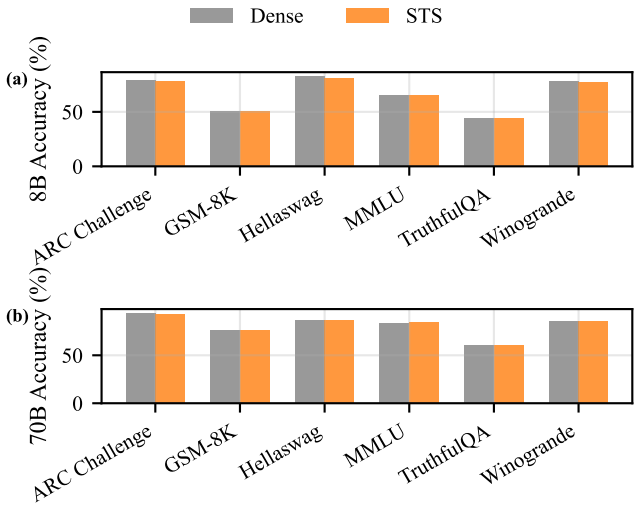


Figure 7: **[General Benchmark Fidelity]: Zero-shot accuracy on standard reasoning benchmarks.** STS incurs negligible accuracy loss across 8B and 70B models, demonstrating that sparsity patterns generalize effectively across model scales.

of memory transfer latency, making CPU offloading a viable option for serving long-context workloads when GPU memory is limited. From a resource utilization perspective, this strategy effectively converts the idle PCIe bandwidth during the computation phase into useful data throughput. By saturating the bus asynchronously, we prevent the I/O subsystem from becoming the critical path, achieving a "compute-bound" rather than "memory-bound" execution profile.

6.2.3 Scalability and Overhead Analysis

To assess the system's robustness in real-world serving scenarios, we evaluate the throughput scalability of STS under varying concurrency levels.

Draft Model Efficiency. A potential concern with draft-based speculation is the computational overhead of the draft model itself. Figure 6 dispels this concern by comparing the throughput of the draft model (Llama-3.2-1B) against the target models (Llama-3.1-8B/70B). The draft model achieves orders of magnitude higher throughput (e.g., $>$ 6000 tokens/s at concurrency 128). This extreme speed ensures that the mask generation step accounts for a trivial fraction of the total end-to-end latency.

Scalability under Load. As concurrency increases from 8 to 128, the draft model exhibits linear scalability without saturation. This indicates that STS is well-suited for high-throughput serving environments, where the draft model can efficiently service multiple target model instances without becoming a bottleneck.

6.3 Fidelity and Robustness

Speed is irrelevant if it is at the cost of destroying the model's accuracy. We extensively verify that STS maintains the comparable quality of the dense baseline. Before analyzing the results, we define the sparsity scopes and granularity configurations used in our evaluation:

- **Sparse Decode (STS-D) vs. Sparse Prefill (STS-P).** We evaluate two configurations: (1) **STS-D**, where the full KV cache is preserved during the prefill phase, and sparsity is applied only during token generation; and (2) **STS-P**, a stricter setting where the prompt itself is compressed on-the-fly, retaining only critical blocks in memory. STS-P serves as a stress test for the draft model's ability to identify semantic importance without hindsight.
- **Page Size Granularity (p_s).** Since vLLM and FlashInfer manage memory in paged blocks, Page Size (p_s) is a critical factor. A smaller p_s allows for finer-grained selection but incurs higher indexing overhead.

6.3.1 Standard & Long-Context Benchmarks

General Reasoning (8B & 70B). Figure 7 shows zero-shot accuracy on benchmarks like MMLU [11] (knowledge),

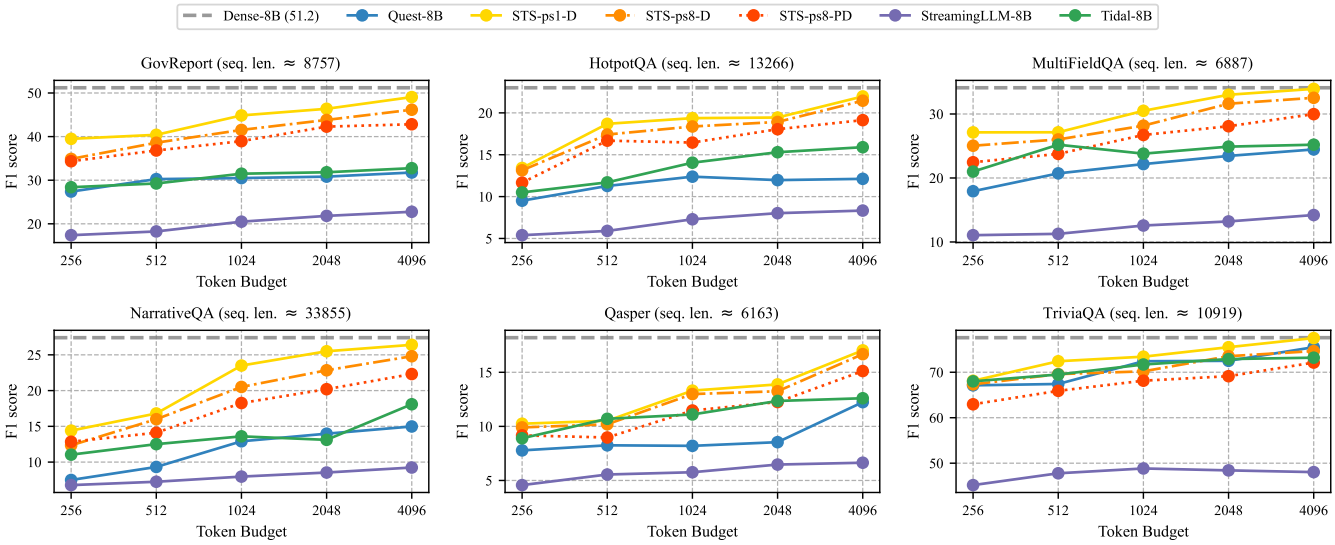


Figure 8: **[LongBench Fidelity]: LongBench results on Llama-3.1-8B.** STS (red) consistently outperforms sparse baselines (Quest, Tidal) and matches Dense performance across diverse tasks.

GSM8K [7] (math), and ARC [6] (reasoning). STS tracks the Dense baseline within 0.5% variance across all tasks for both 8B and 70B models. *Implication:* This result is significant because it proves that the sparsity patterns learned by a small 1B model are semantically aligned with the reasoning pathways of a 70B model. The "critical thinking" tokens are shared across model scales.

Long-Context Robustness & Efficiency. We evaluate performance on LongBench [1] across Llama-3-8B (Figure 8) and Llama-3-70B (Figure 9). STS demonstrates superior robustness compared to baselines, maintaining high fidelity even under constrained conditions.

- **Resilience at Low Budgets:** As shown in the figures, heuristic baselines like Quest and Tidal suffer sharp performance drops when the token budget is restricted (e.g., 256 or 512 tokens in HotpotQA). In contrast, STS maintains a much flatter degradation curve. This indicates that our draft-based selection effectively identifies the "Pareto set" of critical tokens, preserving essential information where other methods fail.

- **Granularity & Prefill Viability:** Crucially, our performance advantage does not rely on expensive fine-grained access. The results show that the coarse-grained configuration ($P_s = 8$, blue line) retains nearly 98% of the performance of the fine-grained setting ($P_s = 1$, red line). This confirms that semantic information is locally clustered, allowing us to exploit hardware-friendly block-sparse kernels. Furthermore, the method remains effective even in the stricter **STS-PD** setting (sparse prefill, green line), where non-essential tokens are pruned during prompt processing. The fact that STS-PD performs comparably to the standard decode-only sparsity suggests that STS can safely compress prompts on-the-fly without losing context.

6.3.2 Synthetic Probes: PPL and Passkey

Perplexity Stability. Figure 10 confirms that STS does not drift as context grows. While TidalDecode shows increasing perplexity degradation ($\Delta > 0.5$) at lengths beyond 8K, STS stays within $\Delta < 0.01$ of the Dense baseline. This stability indicates that STS correctly attends to long-range linguistic dependencies that are crucial for next-token prediction. **Needle-in-a-Haystack Retrieval.** The Passkey task (Figure 11) is a stress test for attention mechanisms. STS achieves **>98% recall** with a budget of just 256 tokens out of 10K. In comparison, heuristic methods often discard the passkey because it appears in a "low-entropy" position. The draft model, however, recognizes the semantic anomaly of the passkey and flags it for the target model to attend to. This phenomenon aligns with the information-theoretic concept of "surprisal." A random passkey inserted into coherent text disrupts the statistical flow, creating a high-entropy signal. Even a small 1B model is sensitive to such disruptions and assigns high attention scores to these anomalies, effectively acting as a "sieve" that catches irregularities for the larger model to process.

6.4 Micro-Analysis: Layer-wise Sparsity

To understand the source of STS 's efficiency and efficacy, we perform a detailed analysis of the sparsity patterns it generates. Figure 12 plots the **Oracle Prunable Ratio** alongside the mask generated by our method. We define this Oracle metric as the theoretical upper bound of sparsity: for each

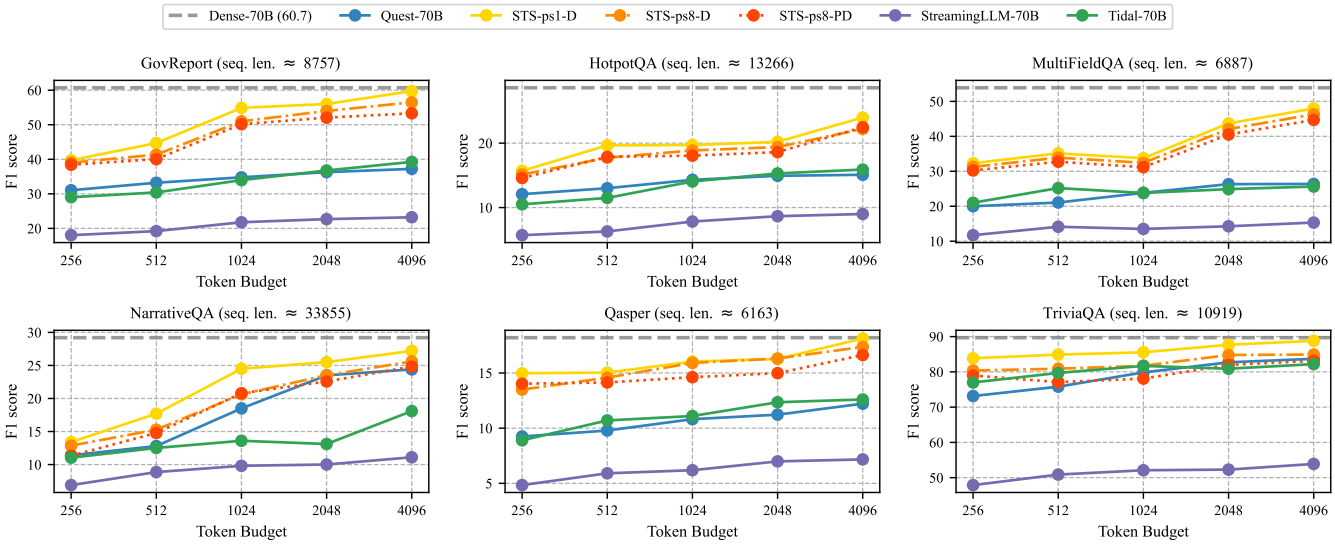


Figure 9: **[LongBench Fidelity]: LongBench results on Llama-3.1-70B.** STS scales effectively to larger models, preserving long-range dependencies even in multi-document QA tasks.

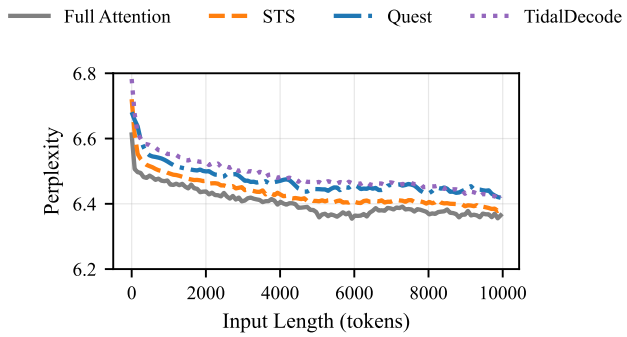


Figure 10: **[Fidelity]: Language modeling perplexity on Wiki-103.** STS maintains perplexity within 0.01 of the dense baseline, whereas heuristic methods degrade at long contexts.

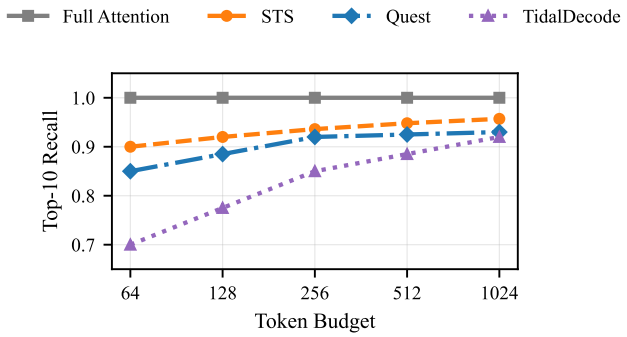


Figure 11: **[Fidelity]: Passkey retrieval recall.** STS maintains >98% recall even at high sparsity levels, proving effective preservation of critical evidence tokens.

layer, it represents the maximum fraction of attention entries that can be discarded without exceeding a strict perplexity degradation threshold ($\Delta\text{PPL} < 0.01$). This sensitivity analysis reveals the inherent *heterogeneity* of the model: some layers are "information-dense" and require full attention, while others are highly redundant.

We observe a distinct **"inverted-U"** sparsity pattern across layers:

- **Initial Layers (Dense):** The first 2 layers require high density (only 5–10% sparsity). This aligns with the *Attention Sink* hypothesis [32], suggesting that initial tokens serve as anchors for global context aggregation. Furthermore, these layers are responsible for low-level feature extraction, requiring broad access to local neighbors to form basic semantic representations.
- **Middle Layers (Highly Sparse):** Layers 3–28 tolerate extreme sparsity (85–95%). These layers typically process local syntax or perform "associative memory" lookups. As noted in induction head theory [24], such operations are highly selective, focusing on only a few specific tokens (e.g., copying from previous occurrences) while ignoring the vast majority of the context.
- **Final Layers (Moderate):** The last few layers exhibit a drop in sparsity (retaining ∼30% of tokens). This indicates a need to synthesize broader context features and calibrate probability distributions for the final next-token prediction, requiring a more holistic view of the sequence.

Crucially, the sparsity mask generated by the draft model (STS) closely tracks this Oracle curve. This confirms our core hypothesis: *small language models share the same fundamen-*

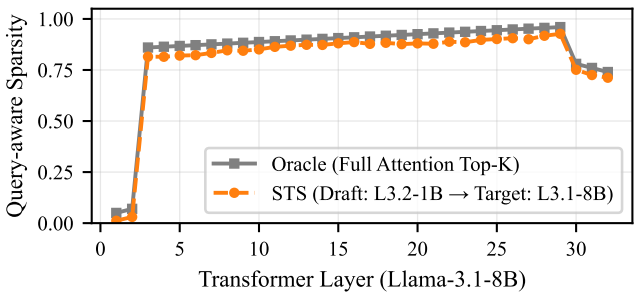


Figure 12: **[Micro-analysis]: Layer-wise sparsity patterns.** The draft-guided mask (STS) closely aligns with Oracle's distribution.

tal attention topology as large models, making them excellent predictors for sparse attention without the need for expensive gradients or full-model computations.

7 Related Work

We classify efficient long-context inference into three paradigms: state eviction (KV compression), heuristic approximation (sparse attention), and speculation-assisted systems.

7.1 KV Cache Compression & Eviction

Systems like H2O [39], StreamingLLM [32], SnapKV [18], and PyramidInfer [33] alleviate memory pressure by permanently evicting tokens based on heuristics such as attention scores or positional recency. While these strategies effectively exploit the sparsity of attention matrices, they rely on the assumption that past token importance predicts future relevance. Consequently, they enforce permanent state eviction, causing irreversible information loss. This is catastrophic for agentic workflows where historical context—often located in the "long tail" of the attention distribution—becomes relevant unexpectedly during multi-step reasoning.

Alternatively, memory-augmented approaches like InfLLM [30] offload KV blocks to CPU memory or disk to avoid data loss. Although this preserves the full context, the limited bandwidth of PCIe interconnects introduces significant data movement overhead. The high latency of retrieving these blocks during the decoding phase creates a bottleneck, preventing the system from meeting the strict latency requirements of real-time interaction.

Unlike these approaches, STS offers the capability to maintain a lossless, GPU-resident context by decoupling storage from computation, ensuring both data integrity and high-throughput access.

7.2 Sparse Attention

To reduce $O(N^2)$ complexity, existing approaches generally fall into static, dynamic, or learned categories. Static methods (e.g., MInference [12], DuoAttention [31]) rely on pre-defined patterns, while dynamic systems (e.g., Quest [26], TidalDecode [34]) estimate token importance on-the-fly. However, dynamic heuristics often incur synchronization overheads on the critical path, and simple metrics may miss semantically subtle "needles."

More recently, SeerAttention [9] introduces a learnable gating mechanism, distilled from dense attention maps to predict block sparsity. While effective, it imposes a significant deployment barrier: it necessitates an additional training phase (e.g., on 0.5B tokens) and structural modifications to the model architecture. Furthermore, SeerAttention primarily targets prefill acceleration, leaving the decoding phase dense.

In contrast, STS is a **training-free** solution that requires no architectural changes or distillation. It functions as a plug-and-play overlay compatible with existing inference engines (e.g., vLLM), leveraging the draft model to achieve zero-overhead sparsification for both prefill and decoding phases.

7.3 Speculative Execution and Pipelining

Speculative decoding [3, 17] originally utilized small draft models solely to accelerate token generation. Recent works have extended this paradigm to context pruning. For instance, LazyLLM [8] progressively selects tokens required for the next step. However, such methods operate on a Just-In-Time (JIT) basis: the selection logic is interleaved with the target model's execution, forcing synchronous waits and preventing kernel overlapping. STS re-architects this into an asynchronous lookahead pipeline. By shifting the burden of sparsity planning entirely to the draft phase, we decouple mask generation from target execution. This achieves the "known-in-advance" property, enabling system-level optimizations—such as latency hiding via prefetching and pipelined memory accesses—that are unattainable in synchronous JIT designs.

8 Conclusion

The quadratic complexity of attention is a critical bottleneck for long-context LLM inference. We introduced STS, a training-free sparse attention mechanism that repurposes the draft model in speculative decoding to generate predictive sparsity masks for the target model. This approach preserves the full KV cache, which is crucial for model fidelity. Our evaluation shows that STS achieves a superior accuracy-sparsity trade-off, maintaining near-lossless performance on language modeling and long-context reasoning benchmarks. By supporting both prefill and decode sparsity, STS is uniquely suited for efficient, multi-turn agentic workloads.

A key contribution of STS is that its sparsity masks are "known-in-advance," enabling novel system-level optimizations like proactive memory prefetching that are infeasible with just-in-time methods. STS thus represents a significant step towards practical and efficient long-context inference, establishing a new state-of-the-art for training-free sparse attention.

References

[1] 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. Longbench: A bilingual, multitask benchmark for long context understanding, 2024.

[2] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.

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

[4] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019.

[5] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking attention with performers, 2022.

[6] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge, 2018.

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

[8] Qichen Fu, Minsik Cho, Thomas Merth, Sachin Mehta, Mohammad Rastegari, and Mahyar Najibi. Lazyllm: Dynamic token pruning for efficient long context llm inference, 2024.

[9] Yizhao Gao, Zhichen Zeng, Dayou Du, Shijie Cao, Peiyuan Zhou, Jiaxing Qi, Junjie Lai, Hayden Kwok-Hay So, Ting Cao, Fan Yang, and Mao Yang. Seerattention: Learning intrinsic sparse attention in your llms, 2025.

[10] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024.

[11] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2021.

[12] Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Minference 1.0: Accelerating prefilling for long-context llms via dynamic sparse attention, 2024.

[13] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024.

[14] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer, 2020.

[15] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

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

[17] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding, 2023.

[18] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation, 2024.

[19] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023.

[20] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents, 2025.

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

[22] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. Deja vu: Contextual sparsity for efficient llms at inference time, 2023.

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

[24] Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Scott Johnston, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. In-context learning and induction heads, 2022.

[25] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.

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

[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[28] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity, 2020.

[29] Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. Mint: Evaluating llms in multi-turn interaction with tools and language feedback, 2024.

[30] Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. Infllm: Training-free long-context extrapolation for llms with an efficient context memory, 2024.

[31] Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. Duoattention: Efficient long-context llm inference with retrieval and streaming heads, 2024.

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

[33] Dongjie Yang, XiaoDong Han, Yan Gao, Yao Hu, Shilin Zhang, and Hai Zhao. Pyramidinfer: Pyramid kv cache compression for high-throughput llm inference, 2024.

[34] Lijie Yang, Zhihao Zhang, Zhuofu Chen, Zikun Li, and Zhihao Jia. Tidaldecode: Fast and accurate llm decoding with position persistent sparse attention, 2024.

[35] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

[36] Zihao Ye et al. Flashinfer: Kernel library for llm serving. https://github.com/flashinfer-ai/flashinfer, 2024.

[37] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Y. X. Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention, 2025.

[38] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences, 2021.

[39] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H_2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023.