SlimInfer: Accelerating Long-Context LLM Inference via Dynamic Token Pruning

Lingkun Long[1], **Ruibin Yang**[1], **Yushi Huang**[2], **Desheng Hui**[1], **Ao Zhou**[1], **Jianlei Yang**[1]*

[1] Beihang University
[2]Hong Kong University of Science and Technology

Abstract

Long-context inference for Large Language Models (LLMs) is heavily limited by high computational demands. While several existing methods optimize attention computation, they still process the full set of hidden states at each layer, limiting overall efficiency. In this work, we propose SlimInfer, an innovative framework that aims to accelerate inference by directly pruning less critical prompt tokens during the forward pass. Our key insight is an *information diffusion phenomenon*: As information from critical tokens propagates through layers, it becomes distributed across the entire sequence. This diffusion process suggests that LLMs can maintain their semantic integrity when excessive tokens, even including these critical ones, are pruned in hidden states. Motivated by this, SlimInfer introduces a dynamic fine-grained pruning mechanism that accurately removes redundant tokens of hidden state at intermediate layers. This layer-wise pruning naturally enables an asynchronous KV cache manager that prefetches required token blocks without complex predictors, reducing both memory usage and I/O costs. Extensive experiments show that SlimInfer can achieve up to $\mathbf{2.53}\times$ time-to-first-token (TTFT) speedup and $\mathbf{1.88}\times$ end-to-end latency reduction for LLaMA3.1-8B-Instruct on a single RTX 4090, without sacrificing performance on LongBench. *Our code will be released upon acceptance.*

1 Introduction

Large Language Models (LLMs) have shown strong performance in long-context tasks such as summarization (Zhang et al. 2020; Kryściński et al. 2022), multi-document question answering (Yang et al. 2018), and retrieval from extended inputs (Bai et al. 2024). Scaling to longer sequences not only enables more complex reasoning, but also introduces substantial computational and memory overhead as the context length increases (Fu 2024).

During the prefill stage, the self-attention mechanism (Vaswani et al. 2017) incurs quadratic time complexity with respect to the sequence length, making it a major source of latency in long-context scenarios. At the same time, the Key-Value (KV) cache grows linearly with input length, leading to substantial GPU memory consumption. To mitigate these issues, numerous token pruning methods have been proposed. However, existing token pruning methods face several critical limitations. Some works (Zhang et al. 2023; Xiao et al. 2024b; Li et al. 2024; Yang et al. 2024; Wang et al. 2025; Cai et al. 2025; Hao et al. 2025; Nguyen et al. 2025) focus primarily on optimizing the decoding phase, offering minimal improvements to the critical Time-To-First-Token (TTFT). In addition, their token eviction strategies often lead to accuracy degradation due to the removal of contextually important information. Other approaches (Lai et al. 2025; Jiang et al. 2024) extend support to both prefill and decoding phases by sparsifying the attention pattern (Deng et al. 2025). Nevertheless, they process the full sequence of hidden states at every layer, leaving non-attention components like Feed-Forward Networks (FFNs) unoptimized and limiting overall acceleration. Memory efficiency presents an additional challenge. Dynamic token pruning methods (Fu et al. 2024) retain the entire KV cache in the GPU, leading to excessive memory consumption and limited scalability for longer sequences. To alleviate this, some systems offload the KV cache to the CPU (Tang et al. 2024), which reduces GPU pressure (Gong et al. 2024; Huang et al. 2024) but introduces significant I/O latency. More recent designs attempt to prefetch KV segments to overlap data transfer and computation (Lee et al. 2024; Yang et al. 2025). However, these approaches often rely on predictor-based mechanisms, introducing additional overhead and complexity. Therefore, existing token pruning methods still struggle to simultaneously optimize inference speed (Jiang et al. 2024; Huang et al. 2025c), memory usage (Xiao et al. 2024b; Huang et al. 2025a), and model performance (Huang et al. 2025b; Wnag et al. 2024).

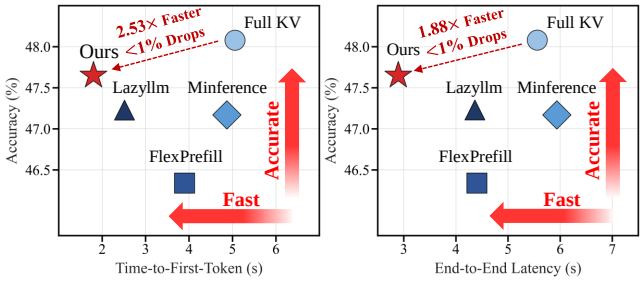

Figure 1: Accuracy *vs.* inference efficiency across different acceleration approaches on LongBench (Bai et al. 2024) for LLaMA-3.1-8B-Instruct (Grattafiori et al. 2024).

*Corresponding author. Email: jianlei@buaa.edu.cn

In this paper, we propose **SlimInfer**, a framework designed to accelerate inference by dynamically pruning less critical prompt tokens during the forward pass. Our method builds on a key insight we term the *information diffusion phenomenon*: As information from critical tokens propagates through the layers of an LLM, it becomes progressively distributed across other token representations. This diffusion process suggests that LLMs can maintain their semantic integrity even when excessive tokens are pruned in hidden states, including those essential initially. Motivated by this insight, SlimInfer introduces a dynamic layer-wise pruning to the hidden states across intermediate layers, progressively reducing computational workload. To preserve essential semantic information while maximising efficiency, we further introduce a fine-grained, block-wise importance evaluation that retains only the contextually relevant tokens. This pruning mechanism works in tandem with an asynchronous KV cache manager, which exploits the determinism of pruning decisions to enable predictor-free prefetching and efficient GPU memory management.

We conduct extensive experiments on LLaMA-3.1-8B-Instruct (Grattafiori et al. 2024) and Qwen2.5-7B-Instruct (Qwen et al. 2025). As shown in Figure 1, SlimInfer can achieve up to $2.53\times$ time-to-first-token (TTFT) speedup and a $1.88\times$ end-to-end latency reduction on a single NVIDIA RTX 4090 GPU. It also maintains near-lossless accuracy drops on the LongBench (Bai et al. 2024).

2 Related Works

2.1 Token Pruning

Token pruning methods aim to reduce inference overhead by selectively removing less critical tokens from computation or memory. Many approaches target GPU memory reduction by maintaining a fixed-size KV cache. StreamingLLM (Xiao et al. 2024b) retains initial tokens (attention sinks) and a sliding window of recent tokens, but discards intermediate ones. H2O (Zhang et al. 2023) proposes a heavy-hitter oracle that evicts tokens with low cumulative attention scores. Similarly, SnapKV (Li et al. 2024) uses the local context of a prompt to predict and retain important tokens for future generation steps. LazyLLM (Fu et al. 2024) introduces dynamic pruning based on token importance, but still retains most KV entries in GPU memory, limiting its scalability to longer contexts. A primary limitation of these methods is their irreversible token eviction, which permanently removes KV entries from GPU memory. This permanent removal can lead to significant accuracy degradation, particularly in complex tasks that rely on long-range dependencies scattered throughout the context. Unlike prior methods that irreversibly discard evicted tokens, SlimInfer offloads currently irrelevant tokens (*i.e.*, pruned tokens) to CPU memory instead of discarding them, significantly improving performance and reducing GPU memory usage. Other methods focus on accelerating computation by inducing sparsity in the attention map. FlexPrefill (Lai et al. 2025), SpargeAttn (Zhang et al. 2025b) adopt block-level heuristics by constructing representative vectors for token chunks, enabling coarse-grained attention skipping. In contrast, MInference (Jiang et al. 2024) predicts structured sparse patterns based on partial attention observations. However, they still compute over the full sequence of hidden states at every layer. As a result, non-attention components like the Feed-Forward Networks (FFN) remain unoptimized, leaving significant room for further acceleration, which limits the overall speedup, especially during the prefill stage. SlimInfer directly addresses this by pruning the hidden states themselves, reducing the workload for all subsequent layers.

2.2 KV Cache Offloading

This line of work addresses the memory overhead of long-context inference by offloading the KV cache from GPU to CPU memory. Quest (Tang et al. 2024) adopts a naive on-demand strategy, which fetches KV entries only when needed. More advanced systems attempt to prefetch KV cache blocks to overlap data transfer with computation. InfiniGen (Lee et al. 2024) performs a lightweight rehearsal using partial model weights and previous-layer inputs, aided by offline Singular Value Decomposition (SVD). AttentionPredictor (Yang et al. 2025) trains a separate CNN to forecast attention scores. However, these approaches introduce considerable computational and engineering overhead. In contrast, SlimInfer sidesteps these limitations by leveraging its layer-wise pruning design to enable a predictor-free prefetching strategy, allowing efficient KV cache transfers without speculative estimation.

3 Motivation

The design of SlimInfer is inspired by the following core insights: (1) *Information diffusion phenomenon*, which confirms the feasibility of aggressive pruning hidden states; (2) This pruning strategy naturally offers an opportunity for KV cache prefetching to further improve inference efficiency.

3.1 Information Diffusion

Conventional token pruning approaches (Lai et al. 2025; Jiang et al. 2024) to accelerating attention computation typically retain the full set of hidden states while optimizing the underlying operations. In contrast, we investigate a more radical direction: The feasibility of pruning hidden states directly during the forward pass. To this end, we conducted a probing experiment on LLaMA3.1-8B-Instruct (Grattafiori et al. 2024). As shown in Figure 2 (left), we selectively remove the hidden state corresponding to a critical prompt token "278" in different layers. The model successfully recalls the correct answer when pruning is applied at a later layer, but fails when pruning occurs earlier. To further understand the underlying mechanism, we visualize the attention weights from the decoding token to the prompt tokens across all transformer layers in Figure 2 (right). In a standard decoding step, a bright vertical activation band emerges around Layer 13, which signifies a sustained focus of the decoding token towards the critical prompt token ("278" in the response $\rightarrow$ "278" in the prompt). When pruning is applied in a later layer (*i.e.*, Layer 25), the activation band is abruptly truncated at the pruning point. Despite this truncation, the model produces the correct output, suggesting that

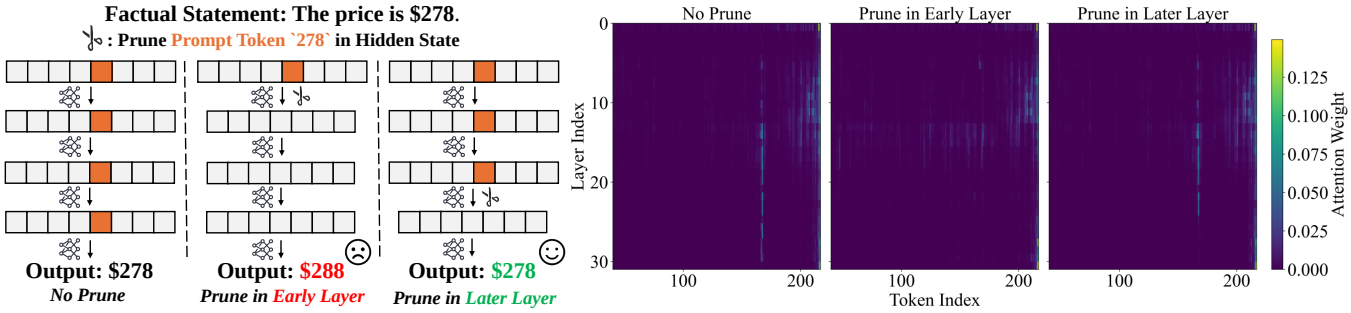

Figure 2: (**Left**) Illustration of a probing experiment on LLaMA3.1-8B-Instruct (Grattafiori et al. 2024). Pruning the hidden state of the critical prompt token "278" (which indicates the correct answer: "$278") in a later layer (right) results in the correct output, whereas pruning prompt tokens in an early layer (middle) leads to an incorrect output. (**Right**) Visualization of layer-wise attention weights from the decoding token (*i.e.*, response token) "278" to prompt tokens.

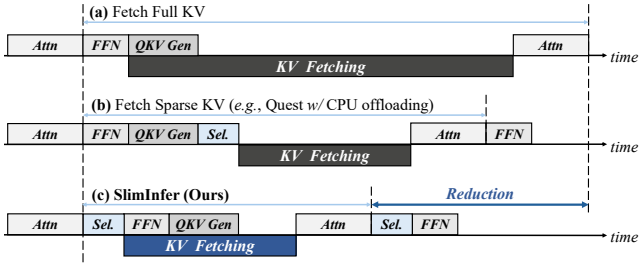

Figure 3: SlimInfer reduces the latency of a layer (*i.e.*, *QKV Generation+ Attention+FFN*) by prefetching KV cache that is offloaded to CPU, overlapping KV cache fetching with computation. "*Sel.*" means selecting tokens in KV cache (b) or hidden state (c) to prune.

the semantic contribution of the critical token has already been effectively diffused into other tokens during the forward passes of the early layers. To the contrary, early pruning at Layer 5 prevents the formation of this stable attention pattern. The absence of the hidden state corresponding to the critical token results in scattered and faint vertical lines over irrelevant tokens around Layer 13. This disoriented attention span reflects a disruption of the standard inference process.

Insights. This series of observations yields two core design principles for SlimInfer: (*i*) The hidden state of the early layer should be retained to preserve semantic fidelity, as pruning too early disrupts the diffusion process; (*ii*) In later layers, even the hidden states of originally important tokens can be safely pruned, indicating substantial redundancy that can be exploited to reduce computation.

3.2 Prefetching Opportunities

Managing the KV cache efficiently is a major challenge in long-context inference, especially when offloading the KV cache to the CPU for GPU memory savings. It introduces significant I/O costs by fetching offloaded KV cache from CPU to GPU during subsequent inference steps (Lee et al. 2024). To reduce this overhead, prior work introduces prefetching, a technique that overlaps KV cache transfer with computation to hide latency. However, enabling prefetching is non-trivial for token pruning that focuses on sparse attention. As shown in Figure 3 (b), Quest (Tang et al. 2024) prunes tokens from the KV cache (including offloaded entries) based on current *QKV* representations. After the pruning stage (*i.e.*, *Sel.*), the offloaded KV entries required for fetching are available. Thus, it is impossible to overlap data transfer (*KV Fetching*) with computation prior to *Attention*. To allow prefetching, InfiniGen (Lee et al. 2024) addresses this by rehearsing attention patterns using partial weights and offline SVD, while AttentionPredictor (Yang et al. 2025) trains a separate CNN to forecast future attention scores. Both approaches introduce additional computational and engineering overhead due to their speculative nature.

Analysis. Notably, with the aforementioned hidden state pruning (Section 3.1) applied following *Attention* for a given layer, SlimInfer can eliminate the need for predictive mechanisms. As illustrated in Figure 3 (c), *KV Fetching* can overlap with the computation of *FFN* and *QKV Generation* prior to the subsequent *Attention*. Building upon this analysis, our framework can naturally achieve timely prefetching without any predictive or heuristic strategy.

4 SlimInfer

4.1 Framework Overview

In this Section, we propose SlimInfer to accelerate long-context inference. It incorporates a dynamic block-wise hidden state pruning with a predictor-free KV cache prefetching strategy. Specifically, the prompt tokens are partitioned into fixed-size blocks, a common abstraction that aligns well with GPU-friendly batch operations and enables efficient memory access (Tang et al. 2024; Xiao et al. 2024a). At any point during inference, a block is called *active block* if it is deemed critical for ongoing computations. Only these active blocks participate in attention computation and have their KV entries stored in GPU memory. Additionally, our pruning mechanism is applied exclusively to prompt tokens. In contrast, all tokens generated as responses is fully retained to preserve fluency throughout generation. As shown in Figure 4, inference is divided into two stages:

Preserve layers. Motivated by our analysis of information diffusion, the early layers retain all tokens in the prompt.

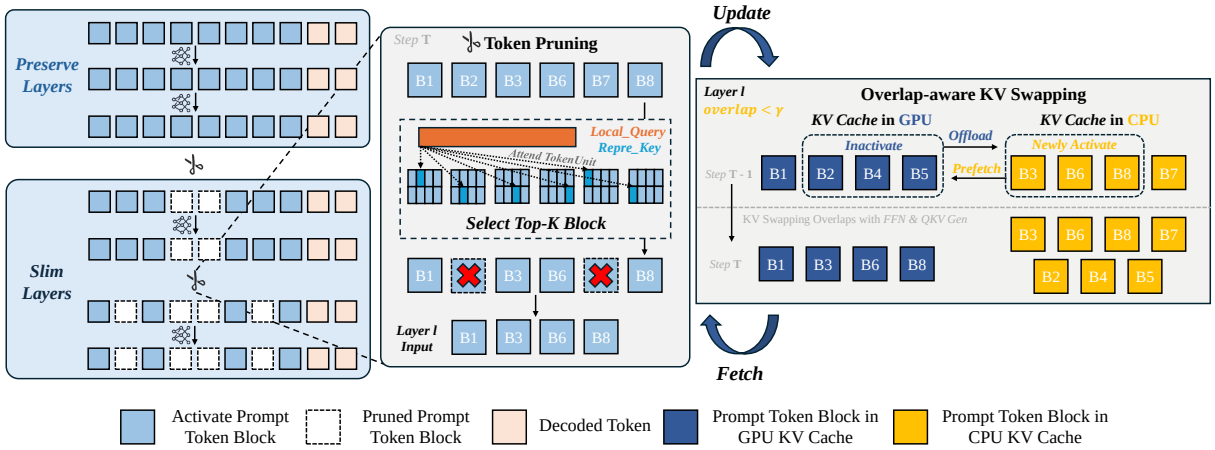

Figure 4: Overview of the proposed **SlimInfer**. (*i*) During inference, early *Preserve Layers* retain all prompt blocks to support *information diffusion* (Section 3.1), while later *Slim Layers* prune less relevant blocks to reduce computation (Section 4.1). (*ii*) Each block is divided into fine-grained *Token Units* for accurate importance scoring (Section 4.2). (*iii*) When $\texttt{overlap} < \gamma$ (Algorithm 1), SlimInfer triggers asynchronous KV cache swapping, which naturally overlaps data transfer (prefetching+offloading) with computation (Section 4.3). "T" denotes the current inference step.

This ensures that critical semantic information has sufficient depth to propagate through the model before pruning.

Slim layers. In the subsequent layers, SlimInfer dynamically prunes prompt blocks of hidden state across inference steps to reduce computation. This is guided by an accurate importance estimator that selects the top-k most relevant blocks based on the recent decoding context. The pruning decisions (as demonstrated in Section 4.2) are made immediately after Attention computation and determine the *active* block set for the next layer. This active block set is then propagated unchanged through subsequent layers until the next pruning operation.

The above hidden state pruning paradigm naturally enables efficient KV cache prefetching, as mentioned in Section 3.2. Moreover, we adopt an overlap-aware design (see Section 4.3) for prefetching to avoid unnecessary data movement: Asynchronous KV cache prefetching and offloading are triggered only when the active block set changes significantly, enabling I/O to be overlapped with computation, thereby minimizing inference overhead.

4.2 Block-wise Prompt Token Pruning

Here we detail the specific pruning decision for *Slim Layers*. Conventional approaches of block-level token pruning often estimate the contribution of each block to the current decoding context by compressing the entire block into a single vector (Tang et al. 2024; Yang et al. 2025), which may obscure fine-grained semantic information. To address this limitation, SlimInfer adopts a more expressive strategy by partitioning each prompt block of Key states into multiple smaller subsets, termed as *token units*. This design captures finer-grained semantics within each block, enabling more accurate importance estimation without sacrificing block-level memory efficiency.

Specifically, each prompt block B_j is divided into M disjoint *TokenUnits*, where each unit consists of a contiguous sequence of tokens within the block. For each token unit, a representative Key vector $k_{\text{rep}}(j,m)$ is computed by averaging the Key states of all tokens within that unit.

$$k_{\text{rep}}(j,m) = \text{Mean}\left(\{\text{key} \in \text{TokenUnit}_{j,m}\}\right). \quad (1)$$

To assess block importance, we construct a local window of the Query state, q_l, by averaging the Query vectors of the most recent w tokens, drawn from the end of the prompt during the prefill phase or from decoded tokens during the decoding phase. For each representative Key vector $k^h_{\text{rep}}(j,m)$ in block B_j, we compute its similarity to q_l via dot product on each attention head. The block-level importance score is then defined as:

$$r_{\text{block}}(q_l, B_j) = \max_m \left\{ \frac{1}{H} \sum_{h=1}^{H} \left(q_l^h \cdot k^h_{\text{rep}}(j,m)\right) \right\}, \quad (2)$$

where H denotes the number of attention heads, m indexes token units within the block, and h indexes attention heads.

In addition to this dynamic scoring, our pruning policy enforces the retention of structurally important blocks to maintain model stability. Specifically, the initial block of the prompt, which often acts as attention sinks (Xiao et al. 2024b), is always preserved in the active set, regardless of its score. For all other blocks, the top-k blocks with the highest importance scores are selected to form the candidate set, $B_{\text{candidate}}(t)$[1], for subsequent computation. The KV cache of pruned blocks is not discarded but is offloaded to CPU memory, allowing future restoration when they become relevant again in subsequent decoding steps.

4.3 Predictor-Free KV Cache Prefetching

To reduce GPU memory pressure, SlimInfer offloads the KV cache of inactive prompt blocks to the CPU. Under

[1] t denotes the current inference step.

Algorithm 1: Overlap-aware KV Swapping

Input: $B_{\text{active}}(t-1)$, $B_{\text{candidate}}(t)$, B_{Memory}, and threshold γ
Output: Updated $B_{\text{active}}(t)$

Compute overlap ratio:

$$\texttt{overlap} \leftarrow \frac{|B_{\text{candidate}}(t) \cap B_{\text{active}}(t-1)|}{|B_{\text{candidate}}(t)|}$$

if $\texttt{overlap} < \gamma$ **then**
  $B_{\text{active}}(t) \leftarrow B_{\text{candidate}}(t)$
  $B_{\text{offload}} \leftarrow (B_{\text{active}}(t-1) \setminus B_{\text{active}}(t)) \setminus B_{\text{Memory}}$
  $B_{\text{load}} \leftarrow (B_{\text{active}}(t) \setminus B_{\text{active}}(t-1))$
  for each block B_j in B_{offload} **do**
    `// Offloading`
    Move KV cache of B_j from GPU to CPU
  end for
  for each block B_j in B_{load} **do**
    `// Prefetching`
    Load KV cache of B_j from CPU to GPU
  end for
else
  $B_{\text{active}}(t) \leftarrow B_{\text{active}}(t-1)$
end if

this scenario, SlimInfer naturally allows a predictor-free prefetching mechanism to reduce significant I/O costs that leverages its layer-wise hidden state pruning design (as demonstrated in Section 3.2). Here, we further present an overlap-aware KV swapping (see Algorithm 1) to minimize unnecessary data transfer for prefetching as follows.

At each inference step $t\,(t > 1)$, SlimInfer maintains an active block set[2], $B_{\text{active}}(t)$, whose corresponding KV cache entries are stored in GPU memory for fast access. To minimize unnecessary data movement, a swap operation (*i.e.*, offloading + prefetching) is only enabled when the composition of this set needs to change significantly. Specifically, SlimInfer first establishes the candidate activation set, $B_{\text{candidate}}(t)$ (see Section 4.2), calculated based on importance scores. SlimInfer then computes the overlap ratio between this candidate set and the previous active block set, $B_{\text{active}}(t-1)$. If the ratio falls below a predefined threshold γ, a swap operation is triggered. Otherwise, $B_{\text{active}}(t-1)$ is directly reused as $B_{\text{active}}(t)$, which neglects KV cache prefetching and incurs negligible performance drops (see Appendix). This design prioritizes inference efficiency to reduce data transfer overhead.

The swap operation, as detailed in Algorithm 1, involves asynchronous prefetching: (*i*) KV entries for newly required blocks (B_{load}) are transferred from the CPU to the GPU. (*ii*) Entries for unneeded blocks (B_{offload}) that are not yet in the CPU memory pool are offloaded to the CPU; those already reside in CPU (*i.e.*, corresponding to blocks B_{Memory}), their GPU memory is immediately released for newly prefetched entries. To maximize efficiency and hide I/O latency, the offloading and prefetching processes are executed on a separate CUDA stream. As illustrated in Figure 3, this swap overlaps with the subsequent *FFN* and *QKV Generation*.

[2]The definition of active block can be found in Section 4.1.

5 Experiments

5.1 Settings

Models The experiments are conducted using LLaMA-3.1-8B-Instruct (LLaMA-3.1) (Grattafiori et al. 2024) and Qwen2.5-7B-Instruct (Qwen-2.5) (Qwen et al. 2025) to evaluate the effectiveness of our method in larger-scale LLMs. Both models support context lengths of 128k.

Implementation Details Our framework is built on LazyLLM (Fu et al. 2024) and is implemented in PyTorch. For the inference pipeline, we integrate SlimInfer into the Transformers (Wolf et al. 2020) library by replacing the default self-attention module to support efficient block-wise token pruning and asynchronous KV cache management. Unless otherwise noted, we use a block size of 64, a token unit size of 8, a KV swap threshold $\gamma = 0.9$, and a local query window of 4. Pruning is applied at layers 10, 20, and 30 for LLaMA3.1, retaining 8k, 4k, and 2k tokens respectively; and at layers 9, 18, and 26 for Qwen2.5, retaining 12k, 6k, and 4k tokens. All accuracy experiments are conducted on an NVIDIA H200 GPU, while efficiency evaluations are run on a single NVIDIA RTX 4090 GPU (24GB) to simulate typical edge deployment.

Baselines To evaluate the effectiveness of **SlimInfer**, we compare it with FlashAttention2 (Full KV) (Dao 2023) and 3 token pruning approaches for long-context processing: MInference (Jiang et al. 2024), FlexPrefill (Lai et al. 2025), and LazyLLM (Fu et al. 2024). FlashAttention2 serves as the dense attention baseline, while the others adopt sparse attention or memory management to improve efficiency. All results are based on public implementations. To ensure a fair comparison, LazyLLM applies pruning at the same layers as SlimInfer, retaining 50% of tokens at each pruning layer. For FlexPrefill, we use $\gamma = 0.95$ for both LLaMA-3.1 and Qwen-2.5, consistent with its recommended configuration. For MInference, we follow its official codebase and select the sparse attention pattern for each head accordingly.

5.2 Accuracy Evaluation

Following common practice (Zhang et al. 2025b; Li et al. 2024; Zhang et al. 2025a), we adopt the LongBench (Bai et al. 2024) to evaluate the generation quality of our method under long-context understanding settings. LongBench includes a wide range of tasks such as single-document and multi-document QA, summarization, few-shot learning, synthetic tasks, and code completion. Each task is evaluated using task-specific metrics such as accuracy, F1-score, and Rouge-L, where higher scores indicate better performance.

As shown in Table 1, SlimInfer consistently achieves the highest average accuracy across both LLaMA3.1-8B-Instruct and Qwen2.5-7B-Instruct models. Beyond its strong overall performance, SlimInfer exhibits consistent and robust accuracy across diverse task categories, matching or surpassing other baselines on most benchmarks. These results underscore its broad generalization capability across different model architectures.

Method	Single-Doc. QA		Multi-Doc. QA			Summarization				Few-shot Learning			Synthetic Task			Code Completion		Avg. (%)
	Qasper	MQA	HPQA	2WiKi	MuSiQue	GovRep	QMSum	MNews	VCSum	TREC	TQA	SAMSum	LSHT	Count	PassR	LCC	RepB-p	
LLaMA3.1-8B-Instruct																		
Full KV	45.82	55.05	55.50	44.28	30.78	35.21	25.49	27.23	17.17	72.50	91.65	43.92	46.00	7.43	99.50	63.12	56.74	48.08
LazyLLM	**46.39**	51.28	54.52	43.42	28.86	34.57	25.41	27.05	17.30	70.50	91.00	43.64	**46.00**	**7.94**	**99.50**	59.44	56.12	47.23
MInference	44.29	52.53	52.00	44.10	25.72	**35.09**	25.47	**27.21**	**17.53**	**72.00**	91.18	43.73	**46.00**	3.25	97.00	**64.87**	60.00	47.17
FlexPrefill	44.55	**55.56**	54.56	43.43	30.07	34.64	**25.83**	27.05	16.97	70.50	89.81	43.18	41.00	2.59	82.00	64.67	**62.06**	46.38
SlimInfer	45.19	53.82	**55.14**	**44.37**	**30.95**	34.99	24.77	27.10	16.81	71.00	**91.65**	**44.36**	45.50	6.30	98.50	63.65	55.95	**47.65**
Qwen2.5-7B-Instruct																		
Full KV	43.92	52.76	57.97	46.56	30.16	31.78	23.36	24.30	16.05	72.50	88.64	45.64	43.00	8.00	100.00	60.44	66.84	47.76
LazyLLM	39.79	45.71	53.30	42.58	28.94	31.16	23.08	23.28	15.61	66.50	87.67	45.31	42.25	6.59	**100.00**	57.46	63.89	45.48
MInference	**44.02**	**52.86**	**58.25**	46.17	**29.85**	31.78	23.27	23.88	15.84	71.50	**89.09**	45.89	41.60	8.00	92.00	**61.33**	**67.98**	47.25
FlexPrefill	41.65	51.92	55.29	41.65	29.69	31.71	23.27	24.05	15.91	70.50	88.22	**46.45**	36.50	2.00	75.00	61.10	63.38	44.61
SlimInfer	43.74	52.31	56.94	**46.62**	27.25	**31.85**	**23.40**	**24.26**	**16.09**	**72.00**	89.01	45.52	**43.00**	**8.50**	99.00	60.21	65.71	**47.38**

Table 1: Performance comparison on LongBench (Bai et al. 2024). The best and second results are in **bold** and underlined.

5.3 Efficiency Evaluation

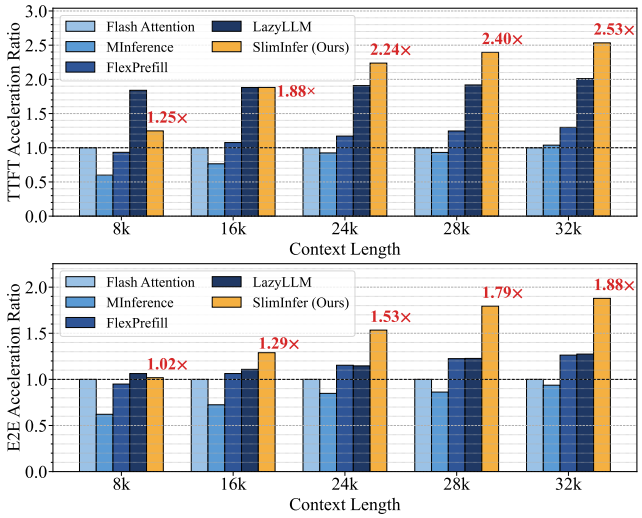

Figure 5: Inference efficiency comparison for LLaMA3.1-8B-Instruct (Grattafiori et al. 2024). (**Upper**) TTFT and (**Lower**) E2E latency acceleration ratio *vs.* context length. SlimInfer far outperforms the baselines at long context lengths ($\geq$24k) and remains on par with them in other cases.

Latency Profiling We benchmark the inference latency across various methods with a single input sequence. All experiments are conducted on an RTX 4090 GPU using LLaMA-3.1-8B-Instruct (Grattafiori et al. 2024). To assess how latency scales with input length, we use 5 truncated versions of a 32k token sequence sampled from LongBench (Bai et al. 2024). We report two metrics: (1) *Time-to-First-Token (TTFT)* latency, and (2) *End-to-End (E2E)* latency for decoding 16 tokens. In Figure 5, we present the acceleration ratios of various inference baselines relative to the FlashAttention-2 (Dao 2023) baseline. Across all input lengths, our method consistently achieves significant speedups in both TTFT and E2E latency. In particular, our SlimiInfer shows an increasing acceleration trend for TTFT as context length grows, highlighting the advantage of our sparse prefill design in long-context scenarios. Compared to other baselines, our method achieves the highest TTFT speedup (up to $\mathbf{2.53}\times$) and E2E speedup (up to $\mathbf{1.88}\times$) at 32k input length. These results further validate the superiority of our design in reducing both prompt prefilling and decoding latency. Comparison for Qwen2.5-7B-Instruct (Qwen et al. 2025) can be found in Appendix.

Accuracy *vs.* Efficiency Our dynamic pruning strategy enables flexible trade-offs between inference efficiency and model accuracy. In Figure 6, we compare end-to-end latency and LongBench (Bai et al. 2024) accuracy across different baselines. The results show that SlimInfer establishes a strong Pareto frontier: It achieves accuracy close to the full KV baseline while substantially reducing latency. Compared to existing methods, SlimInfer offers a more favorable balance between quality and efficiency. Detailed settings are provided in the Appendix.

Memory Efficiency In addition to latency, we evaluate SlimInfer's GPU memory footprint against other representative methods. FlexPrefill (Lai et al. 2025) and MInference (Jiang et al. 2024) optimize computation but retain the full KV cache throughout all layers, resulting in no memory savings. LazyLLM (Fu et al. 2024) applies dynamic pruning but overlooks KV cache offloading, missing an opportunity to reduce substantial GPU memory overhead. In con-

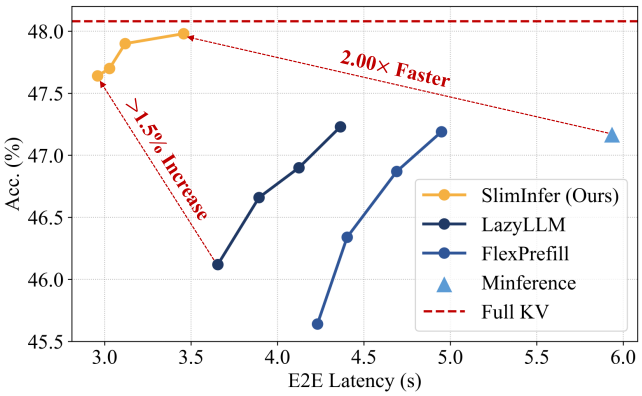

Figure 6: Accuracy *vs.* E2E latency for LLaMA3.1-8B-Instruct (Grattafiori et al. 2024) on LongBench (Bai et al. 2024) with 32k context. SlimInfer achieves a markedly superior trade-off, delivering near-lossless accuracy drops with substantially lower latency than other methods.

Method	8k	16k	24k	28k	32k
Full KV (Baseline)	1.00	2.00	3.00	3.50	4.00
SlimInfer (Ours)	0.80	1.11	1.42	1.58	1.73
Memory Saving (%)	**20.3**	**44.5**	**52.6**	**54.9**	**56.6**

Table 2: Prompt KV cache memory consumption (GB) on LLaMA-3.1-8B-Instruct across different input lengths.

trast, SlimInfer combines a dynamic pruning strategy with offloading for KV pairs from inactive blocks to CPU memory. Therefore, SlimInfer effectively limits GPU memory usage throughout inference. As shown in Table 2, this design yields 20.3−56.6% reductions in prompt KV cache memory.

5.4 Ablation Study

We use LLaMA3.1-8B-Instruct (Grattafiori et al. 2024) here. The default settings are given in Section 5.1.

Balancing Pruning Depth and Token Retention We vary the pruning start layer while keeping the total number of retained tokens approximately constant, to examine how the position of pruning affects model performance across tasks. As shown in Figure 7, all three tasks exhibit a non-

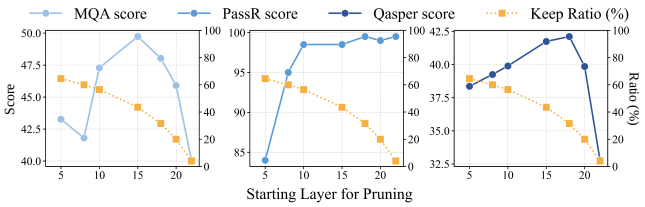

Figure 7: Impact of pruning start layer on task performance under fixed overall sparsity.

linear pattern: Accuracy improves as pruning is delayed to the middle layers. However, for MQA and Qasper, further delaying pruning causes a sharp accuracy drop, while PassR remains largely stable. This could result from too few tokens being retained under the sparsity constraint at later layers. Early pruning hinders *information diffusion*, whereas late pruning restricts token capacity for downstream reasoning. This underscores the need to balance early information preservation with sufficient late-layer token availability.

Method	MuSiQue	PassR	HPQA	Avg. Score
Avg-Pooling	30.52	95.00	55.03	47.45
Max-Pooling	29.77	98.00	54.31	47.44
SlimInfer	**30.95**	**98.50**	**55.14**	**47.65**

Table 3: Ablation study on block importance scoring methods on LongBench (Bai et al. 2024).

Block Importance Scoring Algorithm To assess the effectiveness of our block-wise pruning algorithm, we ablate the block importance scoring strategy. We compare our Token Unit–based method, which partitions each block into finer-grained token units, against two baselines: Avg-Pooling (average of token key states) and Max-Pooling (element-wise maximum). All other SlimInfer settings are kept constant. As shown in Table 3, our approach consistently outperforms the baselines across representative tasks, achieving the highest average score. This highlights the advantage of finer-grained representations in capturing semantic importance for more effective pruning.

Method	8k	16k	24k	28k	32k
Full KV (Baseline)	1.00×	1.00×	1.00×	1.00×	1.00×
Ours (w/o async KV)	1.00×	1.18×	1.37×	1.48×	1.60×
Ours (w/ async KV)	**1.02**×	**1.29**×	**1.53**×	**1.79**×	**1.88**×

Table 4: End-to-end inference speedup across input lengths for LLaMA3.1-8B-Instruct.

Overlapping Operations for Latency Reduction To evaluate the impact of asynchronous KV cache management, we compare end-to-end inference latency with and without this optimization. As shown in Table 4, our method achieves consistent speedups over the FlashAttention baseline across input lengths. At 32k context length, SlimInfer reaches a 1.60× speedup without async KV and further improves to 1.88× with it. The gains increase with input length, demonstrating the effectiveness of overlapping computation and data transfer for long-context inference.

6 Conclusion

We introduce SlimInfer, a framework that accelerates long-context LLM inference through dynamic block-wise token pruning for the hidden state. To preserve essential context, SlimInfer adopts fine-grained importance evaluation to guide accurate and efficient pruning. This deterministic design further supports a predictor-free asynchronous KV cache manager that effectively hides I/O latency. Extensive experiments demonstrate that SlimInfer significantly

improves both Time-To-First-Token and end-to-end latency, without compromising performance.

References

Bai, Y.; Lv, X.; Zhang, J.; Lyu, H.; Tang, J.; Huang, Z.; Du, Z.; Liu, X.; Zeng, A.; Hou, L.; Dong, Y.; Tang, J.; and Li, J. 2024. LongBench: A Bilingual, Multitask Benchmark for Long Context Understanding. arXiv:2308.14508.

Cai, Z.; Zhang, Y.; Gao, B.; Liu, Y.; Li, Y.; Liu, T.; Lu, K.; Xiong, W.; Dong, Y.; Hu, J.; and Xiao, W. 2025. PyramidKV: Dynamic KV Cache Compression based on Pyramidal Information Funneling. arXiv:2406.02069.

Dao, T. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. arXiv:2307.08691.

Deng, Y.; Song, Z.; Xiong, J.; and Yang, C. 2025. How Sparse Attention Approximates Exact Attention? Your Attention is Naturally n^C-Sparse. arXiv:2404.02690.

Fu, Q.; Cho, M.; Merth, T.; Mehta, S.; Rastegari, M.; and Najibi, M. 2024. LazyLLM: Dynamic Token Pruning for Efficient Long Context LLM Inference. arXiv:2407.14057.

Fu, Y. 2024. Challenges in Deploying Long-Context Transformers: A Theoretical Peak Performance Analysis. arXiv:2405.08944.

Gong, R.; Yong, Y.; Gu, S.; Huang, Y.; Lv, C.; Zhang, Y.; Liu, X.; and Tao, D. 2024. LLMC: Benchmarking Large Language Model Quantization with a Versatile Compression Toolkit. arXiv:2405.06001.

Grattafiori, A.; Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Vaughan, A.; Yang, A.; Fan, A.; Goyal, A.; Hartshorn, A.; Yang, A.; Mitra, A.; Sravankumar, A.; Korenev, A.; Hinsvark, A.; Rao, A.; Zhang, A.; Rodriguez, A.; Gregerson, A.; Spataru, A.; Roziere, B.; Biron, B.; Tang, B.; Chern, B.; Caucheteux, C.; Nayak, C.; Bi, C.; Marra, C.; McConnell, C.; Keller, C.; Touret, C.; Wu, C.; Wong, C.; Ferrer, C. C.; Nikolaidis, C.; Allonsius, D.; Song, D.; Pintz, D.; Livshits, D.; Wyatt, D.; Esiobu, D.; Choudhary, D.; Mahajan, D.; Garcia-Olano, D.; Perino, D.; Hupkes, D.; Lakomkin, E.; AlBadawy, E.; Lobanova, E.; Dinan, E.; Smith, E. M.; Radenovic, F.; Guzmán, F.; Zhang, F.; Synnaeve, G.; Lee, G.; Anderson, G. L.; Thattai, G.; Nail, G.; Mialon, G.; Pang, G.; Cucurell, G.; Nguyen, H.; Korevaar, H.; Xu, H.; Touvron, H.; Zarov, I.; Ibarra, I. A.; Kloumann, I.; Misra, I.; Evtimov, I.; Zhang, J.; Copet, J.; Lee, J.; Geffert, J.; Vranes, J.; Park, J.; Mahadeokar, J.; Shah, J.; van der Linde, J.; Billock, J.; Hong, J.; Lee, J.; Fu, J.; Chi, J.; Huang, J.; Liu, J.; Wang, J.; Yu, J.; Bitton, J.; Spisak, J.; Park, J.; Rocca, J.; Johnstun, J.; Saxe, J.; Jia, J.; Alwala, K. V.; Prasad, K.; Upasani, K.; Plawiak, K.; Li, K.; Heafield, K.; Stone, K.; El-Arini, K.; Iyer, K.; Malik, K.; Chiu, K.; Bhalla, K.; Lakhotia, K.; Rantala-Yeary, L.; van der Maaten, L.; Chen, L.; Tan, L.; Jenkins, L.; Martin, L.; Madaan, L.; Malo, L.; Blecher, L.; Landzaat, L.; de Oliveira, L.; Muzzi, M.; Pasupuleti, M.; Singh, M.; Paluri, M.; Kardas, M.; Tsimpoukelli, M.; Oldham, M.; Rita, M.; Pavlova, M.; Kambadur, M.; Lewis, M.; Si, M.; Singh, M. K.; Hassan, M.; Goyal, N.; Torabi, N.; Bashlykov, N.; Bogoychev, N.; Chatterji, N.; Zhang, N.; Duchenne, O.; Çelebi, O.; Alrassy, P.; Zhang, P.; Li, P.; Vasic, P.; Weng, P.; Bhargava, P.; Dubal, P.; Krishnan, P.; Koura, P. S.; Xu, P.; He, Q.; Dong, Q.; Srinivasan, R.; Ganapathy, R.; Calderer, R.; Cabral, R. S.; Stojnic, R.; Raileanu, R.; Maheswari, R.; Girdhar, R.; Patel, R.; Sauvestre, R.; Polidoro, R.; Sumbaly, R.; Taylor, R.; Silva, R.; Hou, R.; Wang, R.; Hosseini, S.; Chennabasappa, S.; Singh, S.; Bell, S.; Kim, S. S.; Edunov, S.; Nie, S.; Narang, S.; Raparthy, S.; Shen, S.; Wan, S.; Bhosale, S.; Zhang, S.; Vandenhende, S.; Batra, S.; Whitman, S.; Sootla, S.; Collot, S.; Gururangan, S.; Borodinsky, S.; Herman, T.; Fowler, T.; Sheasha, T.; Georgiou, T.; Scialom, T.; Speckbacher, T.; Mihaylov, T.; Xiao, T.; Karn, U.; Goswami, V.; Gupta, V.; Ramanathan, V.; Kerkez, V.; Gonguet, V.; Do, V.; Vogeti, V.; Albiero, V.; Petrovic, V.; Chu, W.; Xiong, W.; Fu, W.; Meers, W.; Martinet, X.; Wang, X.; Wang, X.; Tan, X. E.; Xia, X.; Xie, X.; Jia, X.; Wang, X.; Goldschlag, Y.; Gaur, Y.; Babaei, Y.; Wen, Y.; Song, Y.; Zhang, Y.; Li, Y.; Mao, Y.; Coudert, Z. D.; Yan, Z.; Chen, Z.; Papakipos, Z.; Singh, A.; Srivastava, A.; Jain, A.; Kelsey, A.; Shajnfeld, A.; Gangidi, A.; Victoria, A.; Goldstand, A.; Menon, A.; Sharma, A.; Boesenberg, A.; Baevski, A.; Feinstein, A.; Kallet, A.; Sangani, A.; Teo, A.; Yunus, A.; Lupu, A.; Alvarado, A.; Caples, A.; Gu, A.; Ho, A.; Poulton, A.; Ryan, A.; Ramchandani, A.; Dong, A.; Franco, A.; Goyal, A.; Saraf, A.; Chowdhury, A.; Gabriel, A.; Bharambe, A.; Eisenman, A.; Yazdan, A.; James, B.; Maurer, B.; Leonhardi, B.; Huang, B.; Loyd, B.; Paola, B. D.; Paranjape, B.; Liu, B.; Wu, B.; Ni, B.; Hancock, B.; Wasti, B.; Spence, B.; Stojkovic, B.; Gamido, B.; Montalvo, B.; Parker, C.; Burton, C.; Mejia, C.; Liu, C.; Wang, C.; Kim, C.; Zhou, C.; Hu, C.; Chu, C.-H.; Cai, C.; Tindal, C.; Feichtenhofer, C.; Gao, C.; Civin, D.; Beaty, D.; Kreymer, D.; Li, D.; Adkins, D.; Xu, D.; Testuggine, D.; David, D.; Parikh, D.; Liskovich, D.; Foss, D.; Wang, D.; Le, D.; Holland, D.; Dowling, E.; Jamil, E.; Montgomery, E.; Presani, E.; Hahn, E.; Wood, E.; Le, E.-T.; Brinkman, E.; Arcaute, E.; Dunbar, E.; Smothers, E.; Sun, F.; Kreuk, F.; Tian, F.; Kokkinos, F.; Ozgenel, F.; Caggioni, F.; Kanayet, F.; Seide, F.; Florez, G. M.; Schwarz, G.; Badeer, G.; Swee, G.; Halpern, G.; Herman, G.; Sizov, G.; Guangyi; Zhang; Lakshminarayanan, G.; Inan, H.; Shojanazeri, H.; Zou, H.; Wang, H.; Zha, H.; Habeeb, H.; Rudolph, H.; Suk, H.; Aspegren, H.; Goldman, H.; Zhan, H.; Damlaj, I.; Molybog, I.; Tufanov, I.; Leontiadis, I.; Veliche, I.-E.; Gat, I.; Weissman, J.; Geboski, J.; Kohli, J.; Lam, J.; Asher, J.; Gaya, J.-B.; Marcus, J.; Tang, J.; Chan, J.; Zhen, J.; Reizenstein, J.; Teboul, J.; Zhong, J.; Jin, J.; Yang, J.; Cummings, J.; Carvill, J.; Shepard, J.; McPhie, J.; Torres, J.; Ginsburg, J.; Wang, J.; Wu, K.; U, K. H.; Saxena, K.; Khandelwal, K.; Zand, K.; Matosich, K.; Veeraraghavan, K.; Michelena, K.; Li, K.; Jagadeesh, K.; Huang, K.; Chawla, K.; Huang, K.; Chen, L.; Garg, L.; A, L.; Silva, L.; Bell, L.; Zhang, L.; Guo, L.; Yu, L.; Moshkovich, L.; Wehrstedt, L.; Khabsa, M.; Avalani, M.; Bhatt, M.; Mankus, M.; Hasson, M.; Lennie, M.; Reso, M.; Groshev, M.; Naumov, M.; Lathi, M.; Keneally, M.; Liu, M.; Seltzer, M. L.; Valko, M.; Restrepo, M.; Patel, M.; Vyatskov, M.; Samvelyan, M.; Clark, M.; Macey, M.; Wang, M.; Hermoso, M. J.; Metanat, M.; Rastegari, M.; Bansal, M.; Santhanam, N.; Parks, N.; White, N.; Bawa, N.; Singhal, N.;

Egebo, N.; Usunier, N.; Mehta, N.; Laptev, N. P.; Dong, N.; Cheng, N.; Chernoguz, O.; Hart, O.; Salpekar, O.; Kalinli, O.; Kent, P.; Parekh, P.; Saab, P.; Balaji, P.; Rittner, P.; Bontrager, P.; Roux, P.; Dollar, P.; Zvyagina, P.; Ratanchandani, P.; Yuvraj, P.; Liang, Q.; Alao, R.; Rodriguez, R.; Ayub, R.; Murthy, R.; Nayani, R.; Mitra, R.; Parthasarathy, R.; Li, R.; Hogan, R.; Battey, R.; Wang, R.; Howes, R.; Rinott, R.; Mehta, S.; Siby, S.; Bondu, S. J.; Datta, S.; Chugh, S.; Hunt, S.; Dhillon, S.; Sidorov, S.; Pan, S.; Mahajan, S.; Verma, S.; Yamamoto, S.; Ramaswamy, S.; Lindsay, S.; Lindsay, S.; Feng, S.; Lin, S.; Zha, S. C.; Patil, S.; Shankar, S.; Zhang, S.; Zhang, S.; Wang, S.; Agarwal, S.; Sajuyigbe, S.; Chintala, S.; Max, S.; Chen, S.; Kehoe, S.; Satterfield, S.; Govindaprasad, S.; Gupta, S.; Deng, S.; Cho, S.; Virk, S.; Subramanian, S.; Choudhury, S.; Goldman, S.; Remez, T.; Glaser, T.; Best, T.; Koehler, T.; Robinson, T.; Li, T.; Zhang, T.; Matthews, T.; Chou, T.; Shaked, T.; Vontimitta, V.; Ajayi, V.; Montanez, V.; Mohan, V.; Kumar, V. S.; Mangla, V.; Ionescu, V.; Poenaru, V.; Mihailescu, V. T.; Ivanov, V.; Li, W.; Wang, W.; Jiang, W.; Bouaziz, W.; Constable, W.; Tang, X.; Wu, X.; Wang, X.; Wu, X.; Gao, X.; Kleinman, Y.; Chen, Y.; Hu, Y.; Jia, Y.; Qi, Y.; Li, Y.; Zhang, Y.; Zhang, Y.; Adi, Y.; Nam, Y.; Yu; Wang; Zhao, Y.; Hao, Y.; Qian, Y.; Li, Y.; He, Y.; Rait, Z.; DeVito, Z.; Rosnbrick, Z.; Wen, Z.; Yang, Z.; Zhao, Z.; and Ma, Z. 2024. The Llama 3 Herd of Models. arXiv:2407.21783.

Hao, J.; Zhu, Y.; Wang, T.; Yu, J.; Xin, X.; Zheng, B.; Ren, Z.; and Guo, S. 2025. OmniKV: Dynamic Context Selection for Efficient Long-Context LLMs. In *The Thirteenth International Conference on Learning Representations*.

Huang, Y.; Gong, R.; Liu, J.; Chen, T.; and Liu, X. 2024. TFMQ-DM: Temporal Feature Maintenance Quantization for Diffusion Models. arXiv:2311.16503.

Huang, Y.; Gong, R.; Liu, J.; Ding, Y.; Lv, C.; Qin, H.; and Zhang, J. 2025a. QVGen: Pushing the Limit of Quantized Video Generative Models. arXiv:2505.11497.

Huang, Y.; Gong, R.; Liu, X.; Liu, J.; Li, Y.; Lu, J.; and Tao, D. 2025b. Temporal Feature Matters: A Framework for Diffusion Model Quantization. arXiv:2407.19547.

Huang, Y.; Wang, Z.; Gong, R.; Liu, J.; Zhang, X.; Guo, J.; Liu, X.; and Zhang, J. 2025c. HarmoniCa: Harmonizing Training and Inference for Better Feature Caching in Diffusion Transformer Acceleration. arXiv:2410.01723.

Jiang, H.; Li, Y.; Zhang, C.; Wu, Q.; Luo, X.; Ahn, S.; Han, Z.; Abdi, A. H.; Li, D.; Lin, C.-Y.; Yang, Y.; and Qiu, L. 2024. MInference 1.0: Accelerating Pre-filling for Long-Context LLMs via Dynamic Sparse Attention. arXiv:2407.02490.

Kamradt, G. 2023. Needle in a Haystack - Pressure Testing LLMs. https://github.com/gkamradt/LLMTest_NeedleInAHaystack.

Kryściński, W.; Rajani, N.; Agarwal, D.; Xiong, C.; and Radev, D. 2022. BookSum: A Collection of Datasets for Long-form Narrative Summarization. arXiv:2105.08209.

Lai, X.; Lu, J.; Luo, Y.; Ma, Y.; and Zhou, X. 2025. FlexPrefill: A Context-Aware Sparse Attention Mechanism for Efficient Long-Sequence Inference. arXiv:2502.20766.

Lee, W.; Lee, J.; Seo, J.; and Sim, J. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. arXiv:2406.19707.

Li, Y.; Huang, Y.; Yang, B.; Venkitesh, B.; Locatelli, A.; Ye, H.; Cai, T.; Lewis, P.; and Chen, D. 2024. SnapKV: LLM Knows What You are Looking for Before Generation. arXiv:2404.14469.

Nguyen, A.; Schafft, S.; Hale, N.; and Alfaro, J. 2025. FASTGEN: Fast and Cost-Effective Synthetic Tabular Data Generation with LLMs. arXiv:2507.15839.

Qwen; :; Yang, A.; Yang, B.; Zhang, B.; Hui, B.; Zheng, B.; Yu, B.; Li, C.; Liu, D.; Huang, F.; Wei, H.; Lin, H.; Yang, J.; Tu, J.; Zhang, J.; Yang, J.; Yang, J.; Zhou, J.; Lin, J.; Dang, K.; Lu, K.; Bao, K.; Yang, K.; Yu, L.; Li, M.; Xue, M.; Zhang, P.; Zhu, Q.; Men, R.; Lin, R.; Li, T.; Tang, T.; Xia, T.; Ren, X.; Ren, X.; Fan, Y.; Su, Y.; Zhang, Y.; Wan, Y.; Liu, Y.; Cui, Z.; Zhang, Z.; and Qiu, Z. 2025. Qwen2.5 Technical Report. arXiv:2412.15115.

Tang, J.; Zhao, Y.; Zhu, K.; Xiao, G.; Kasikci, B.; and Han, S. 2024. Quest: Query-Aware Sparsity for Efficient Long-Context LLM Inference. arXiv:2406.10774.

Tillet, P.; Kung, H. T.; and Cox, D. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, 10–19. New York, NY, USA: Association for Computing Machinery. ISBN 9781450367196.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention Is All You Need. arXiv:1706.03762.

Wang, G.; Upasani, S.; Wu, C.; Gandhi, D.; Li, J.; Hu, C.; Li, B.; and Thakker, U. 2025. LLMs Know What to Drop: Self-Attention Guided KV Cache Eviction for Efficient Long-Context Inference. arXiv:2503.08879.

Wnag, Z.; Guo, J.; Gong, R.; Yong, Y.; Liu, A.; Huang, Y.; Liu, J.; and Liu, X. 2024. PTSBench: A Comprehensive Post-Training Sparsity Benchmark Towards Algorithms and Models. arXiv:2412.07268.

Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; Davison, J.; Shleifer, S.; von Platen, P.; Ma, C.; Jernite, Y.; Plu, J.; Xu, C.; Scao, T. L.; Gugger, S.; Drame, M.; Lhoest, Q.; and Rush, A. M. 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. arXiv:1910.03771.

Xiao, C.; Zhang, P.; Han, X.; Xiao, G.; Lin, Y.; Zhang, Z.; Liu, Z.; and Sun, M. 2024a. InfLLM: Training-Free Long-Context Extrapolation for LLMs with an Efficient Context Memory. arXiv:2402.04617.

Xiao, G.; Tian, Y.; Chen, B.; Han, S.; and Lewis, M. 2024b. Efficient Streaming Language Models with Attention Sinks. arXiv:2309.17453.

Yang, L.; Zhang, Z.; Chen, Z.; Li, Z.; and Jia, Z. 2024. TidalDecode: Fast and Accurate LLM Decoding with Position Persistent Sparse Attention. arXiv:2410.05076.

Yang, Q.; Wang, J.; Li, X.; Wang, Z.; Chen, C.; Chen, L.; Yu, X.; Liu, W.; Hao, J.; Yuan, M.; and Li, B. 2025. AttentionPredictor: Temporal Pattern Matters for Efficient LLM Inference. arXiv:2502.04077.

Yang, Z.; Qi, P.; Zhang, S.; Bengio, Y.; Cohen, W. W.; Salakhutdinov, R.; and Manning, C. D. 2018. HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering. arXiv:1809.09600.

Zhang, H.; Ji, X.; Chen, Y.; Fu, F.; Miao, X.; Nie, X.; Chen, W.; and Cui, B. 2025a. PQCache: Product Quantization-based KVCache for Long Context LLM Inference. arXiv:2407.12820.

Zhang, J.; Xiang, C.; Huang, H.; Wei, J.; Xi, H.; Zhu, J.; and Chen, J. 2025b. SpargeAttention: Accurate and Training-free Sparse Attention Accelerating Any Model Inference. arXiv:2502.18137.

Zhang, J.; Zhao, Y.; Saleh, M.; and Liu, P. J. 2020. PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization. arXiv:1912.08777.

Zhang, Z.; Sheng, Y.; Zhou, T.; Chen, T.; Zheng, L.; Cai, R.; Song, Z.; Tian, Y.; Ré, C.; Barrett, C.; Wang, Z.; and Chen, B. 2023. H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. arXiv:2306.14048.

Appendix

A Additional Accuracy Experiments

A.1 Needle-in-a-Haystack Result

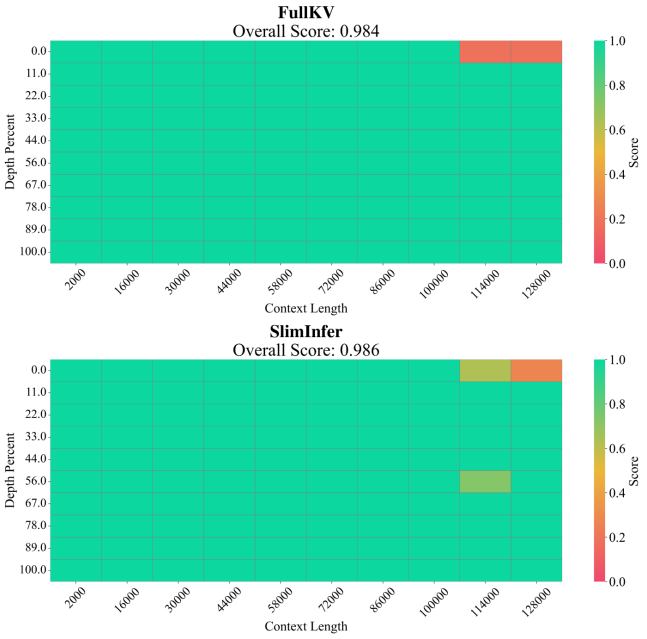

Figure A: Needle-in-a-Haystack (Kamradt 2023) test results for Full KV and SlimInfer on LLaMA3.1-8B-Instruct (Grattafiori et al. 2024). SlimInfer slightly outperforms the Full KV baseline, demonstrating its ability to enhance retrieval accuracy while accelerating inference.

To assess long-context retrieval, we conduct the Needle-in-a-Haystack (Kamradt 2023) test using LLaMA3.1-8B-Instruct (Grattafiori et al. 2024), with SlimInfer pruning 50% of tokens at layers 10, 20, and 30. Other configurations remain the same as in the main text. As shown in Figure A, SlimInfer slightly outperforms the Full KV (Flash Attention2 (Dao 2023)) baseline (0.986 *vs.* 0.984), suggesting that pruning irrelevant tokens reduces noise and helps the model focus on critical information. This demonstrates that SlimInfer not only accelerates inference but can also enhance retrieval accuracy in long contexts.

B Additional Efficiency Experiments

B.1 Qwen2.5 Latency Profiling

In addition to LLaMA3.1-8B-Instruct (Grattafiori et al. 2024), we also evaluate the performance of SlimInfer on Qwen2.5-7B-Instruct (Qwen et al. 2025), following the same experimental setup. The results, presented in Figure B, demonstrate that SlimInfer consistently outperforms the baselines on the Qwen2.5-7B-Instruct model as well. Specifically, SlimInfer achieves up to **2.14**× speedup in Time-to-First-Token (TTFT) and **1.84**× speedup in End-to-End (E2E) latency at a 32k context length. Similar to the

Method		Single-Doc. QA		Multi-Doc. QA			Summarization				Few-shot Learning			Synthetic Task			Code Completion		Avg. (%)	E2E (s)
		Qasper	MQA	HPQA	2WiKi	MuSiQue	GovRep	QMSum	MNews	VCSum	TREC	TQA	SAMSum	LSHT	Count	PassR	LCC	RepB-p		
LLaMA3.1-8B-Instruct																				
Full KV		45.82	55.05	55.50	44.28	30.78	35.21	25.49	27.23	17.17	72.50	91.65	43.92	46.00	7.43	99.50	63.12	56.74	48.08	5.561
MInference		44.29	52.53	52.00	44.10	25.72	35.09	25.47	27.21	17.53	72.00	91.18	43.73	46.00	3.25	97.00	64.87	60.00	47.17	5.935
FlexPrefill	$\gamma = 0.99$	44.06	55.64	55.07	44.91	32.31	35.01	25.22	27.13	17.39	72.00	91.65	43.93	45.50	3.19	85.50	64.29	59.40	47.19	4.949
	$\gamma = 0.98$	44.83	57.31	55.93	42.14	30.95	34.85	24.97	27.13	17.27	69.50	91.64	44.07	45.50	4.31	81.00	64.30	61.06	46.87	4.690
	$\gamma = 0.95$	44.55	55.56	54.56	43.43	30.07	34.64	25.83	27.05	16.97	70.50	89.81	43.18	41.00	2.59	82.00	64.67	62.06	46.38	4.403
	$\gamma = 0.90$	43.64	54.56	55.56	35.74	26.05	34.53	25.07	27.11	17.13	69.00	91.04	42.93	40.00	2.16	82.50	65.56	63.33	45.64	4.231
LazyLLM	50%	46.39	51.28	54.52	43.42	28.86	34.57	25.41	27.05	17.30	70.50	91.00	43.64	46.00	7.94	99.50	59.44	56.12	47.23	4.364
	45%	46.78	53.69	49.34	42.94	28.29	34.39	25.35	26.63	17.26	70.00	91.52	43.94	46.00	7.47	99.50	58.47	55.78	46.90	4.124
	40%	44.31	55.51	49.12	42.61	30.14	33.97	25.33	26.67	17.60	69.00	91.23	43.86	45.50	7.24	99.50	56.28	55.30	46.66	3.893
	35%	42.43	52.76	47.41	43.14	30.74	33.98	24.71	26.66	17.05	69.00	91.23	43.71	46.00	6.33	99.50	55.17	54.14	46.12	3.655
SlimInfer (Ours)	(16, 8, 4)	45.62	55.93	54.86	43.43	30.00	34.87	24.88	27.20	17.58	72.50	91.47	44.31	46.50	6.96	99.50	63.33	56.73	47.98	3.457
	(12, 6, 4)	45.58	53.55	54.76	45.25	30.72	35.15	25.15	27.20	17.42	72.50	91.48	44.27	45.50	6.94	99.00	63.46	56.41	47.90	3.119
	(10, 5, 3)	45.71	53.95	54.45	43.51	29.78	34.97	25.09	27.24	17.12	71.50	91.48	44.33	46.00	6.88	99.00	63.46	56.41	47.70	3.028
	(8, 4, 2)	45.19	53.82	55.14	44.37	30.95	34.99	24.77	27.10	16.81	71.00	91.65	44.36	45.50	6.30	98.50	63.65	55.95	47.65	2.959

Table E: Detailed performance and latency comparison on LongBench (Bai et al. 2024) for LLaMA3.1-8B-Instruct (Grattafiori et al. 2024). We report results for baselines and various configurations of our method, SlimInfer. The numbers in parentheses for SlimInfer, *e.g.*, (16,8,4), denote the number of retained prompt tokens (in units of k, where 1k = 1024) at each of the three pruning layers, respectively. The E2E Latency column shows the time in seconds to generate 16 tokens with a 32k input.

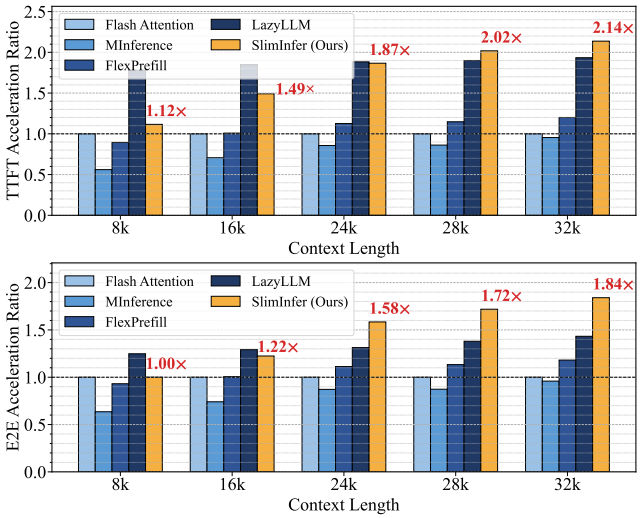

Figure B: Inference efficiency comparison for Qwen2.5-7B-Instruct (Qwen et al. 2025). (**Upper**) TTFT and (**Lower**) E2E latency acceleration ratio *vs.* context length. SlimInfer maintains a significant performance advantage, particularly at longer context lengths.

results on LLaMA3.1, our method shows a growing acceleration trend as the context length increases, further underscoring its effectiveness in long-context scenarios.

B.2 Details for Accuracy *vs.* Efficiency Analysis

This section details the specific configurations for the accuracy *vs.* efficiency trade-off results (*i.e.*, the visualization of Pareto frontier in the main text), as presented comprehensively in Table E. For FlexPrefill, we varied its sparsity threshold γ across $\{0.99, 0.98, 0.95, 0.90\}$. For LazyLLM, we adjusted the token retention ratio through the set $\{50\%, 45\%, 40\%, 35\%\}$. For our method, SlimInfer, we tested four pruning schedules at layers 10, 20, and 30, denoted by the number of retained tokens (in units of k, where 1k = 1024): (16, 8, 4), (12, 6, 4), (10, 5, 3), and (8, 4, 2). The (8, 4, 2) setting represents our most aggressive configuration and corresponds to the main results in the paper. Other baselines like Full KV and MInference were run with their default settings. The results in Table E illustrate the direct relationship between computational reduction and model performance. As expected, more aggressive configurations in FlexPrefill, LazyLLM, and SlimInfer lead to lower E2E latency. Notably, SlimInfer's configurations consistently offer a more favorable balance, achieving substantial latency reductions while incurring minimal accuracy degradation, effectively pushing the Pareto frontier.

B.3 Performance on Edge Devices

To further validate the practicality and broad applicability of SlimInfer, we conducted additional performance evaluations on an edge computing platform, the NVIDIA Jetson AGX Orin (32GB). Due to limitations of the Triton (Tillet, Kung, and Cox 2019), which is not supported on the Jetson plat-

Context	Full KV (Baseline)		SlimInfer (Ours)		Speedup	
	TTFT	E2E	TTFT	E2E	TTFT	E2E
4k	12.49	18.22	12.76	19.56	0.98×	0.93×
8k	26.16	34.51	22.78	33.34	1.15×	1.04×
16k	57.05	72.71	35.31	59.38	1.62×	1.22×
24k	93.31	115.60	51.01	85.62	1.83×	1.35×
28k	113.35	138.62	59.29	86.58	1.91×	1.60×
32k	134.73	163.10	69.57	96.06	**1.94**×	**1.69**×

Table F: Performance comparison on NVIDIA Jetson AGX Orin (32GB) using LLaMA3.1-8B-Instruct (Grattafiori et al. 2024). Latency is measured in seconds. The end-to-end (E2E) latency corresponds to the time taken to generate 16 tokens.

form, we were unable to run other baselines such as MInference and FlexPrefill. Therefore, we directly compare SlimInfer against a highly-optimized Full KV baseline. For this experiment, we used the LLaMA3.1-8B-Instruct model with the (8, 4, 2) pruning configuration for SlimInfer. The results are summarized in Table F. SlimInfer demonstrates substantial performance gains over the Full KV baseline across all context lengths. Notably, at a 32k context length, SlimInfer achieves a **1.94**× speedup in Time-to-First-Token (TTFT) and a **1.69**× speedup in End-to-End (E2E) latency for generating 16 tokens. These results indicate that SlimInfer is not only effective on high-end server GPUs but also provides significant acceleration on edge devices, highlighting its excellent generalization and practical value for real-world deployment.

C Additional Ablation Study

We use LLaMA3.1-8B-Instruct (Grattafiori et al. 2024) here. The default settings are given in the main text.

C.1 Ablation Study on Swap Threshold

Threshold (γ)	Avg. Score (%)	E2E (s)
0.99	47.67	3.057
0.95	47.67	3.040
0.90	47.65	2.959

Table G: Ablation on the swap threshold (γ) for LLaMA3.1-8B-Instruct (Grattafiori et al. 2024). We report the average score on LongBench (Bai et al. 2024) and E2E latency (16 tokens generated, 32k input).

We study the impact of the swapping threshold, γ, which controls the frequency of asynchronous data transfers. A lower γ reduces I/O overhead by allowing more tolerance for changes in the active token set. As shown in Table G, decreasing the threshold from 0.99 to our default value of 0.90 reduces the E2E latency from 3.057s to 2.959s. This performance gain comes at the cost of a negligible 0.02% drop in LongBench (Bai et al. 2024) average score. This favorable trade-off justifies our choice of $\gamma = 0.90$ to optimize for inference speed with minimal impact on accuracy.

C.2 Ablation Study on Block Size

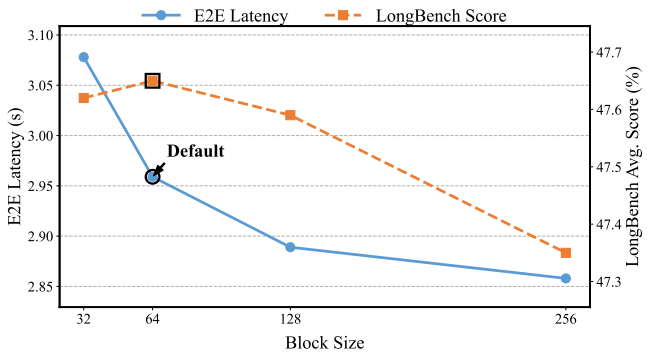

Figure C: Ablation study on block size, showing the LongBench (Bai et al. 2024) average score and end-to-end (E2E) latency under different block size settings.

We study the effect of block size on both performance and accuracy. As shown in Figure C, increasing the block size consistently reduces E2E latency, measured under a 32k token input and 16 token output setting. This reduction stems from improved computational and memory efficiency on the GPU, as larger blocks reduce the number of memory operations and pruning decisions required. However, the impact on accuracy is non-monotonic. When the block size is too small (*e.g.*, 32), each block contains limited context, making it difficult to capture coherent semantic patterns, which undermines the accuracy of importance evaluation. As the block size increases, accuracy improves and peaks at block size 64, where local semantic structure is preserved without introducing too much noise. Beyond this point, further increases in block size lead to performance degradation, as overly large blocks tend to incorporate irrelevant tokens, diluting critical information and reducing pruning precision.

C.3 Ablation Study on Local Query Window Size

We perform an ablation study on the local query window size (w), which determines the number of recent tokens used to construct the local Query vector for importance scoring. This parameter plays a crucial role in accurately identifying relevant prompt blocks based on the current semantic context. As shown in Figure D, the model's performance is highly sensitive to the choice of window size. The average

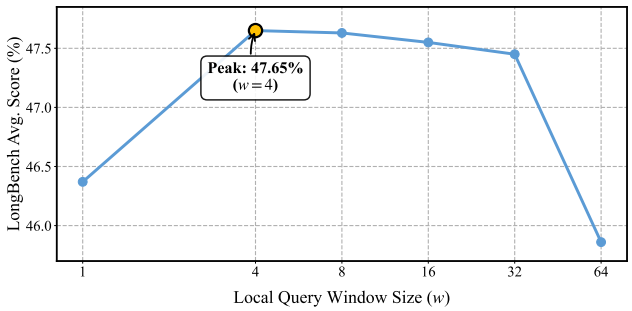

Figure D: Ablation study on the local query window size (w), showing the average score on LongBench (Bai et al. 2024) under different window size settings.

score on LongBench (Bai et al. 2024) reaches its peak at $\mathbf{w = 4}$. A smaller window size (*e.g.*, $w = 1$) results in a notable performance drop, likely due to insufficient contextual information for reliable importance estimation. In contrast, excessively large windows (*e.g.*, $w \geq 8$) also lead to a gradual decrease in precision, suggesting that they may incorporate outdated or irrelevant tokens, thus diluting the effectiveness of the importance score. Overall, a window size of 4 offers an optimal balance by capturing a sufficiently stable and relevant context while minimizing the influence of semantic noise.