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

Lingkun Long[1], **Rubing 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 2.53× time-to-first-token (TTFT) speedup and 1.88× end-to-end latency reduction for LLaMA3.1-8B-Instruct on a single RTX 4090, without sacrificing performance on LongBench.

Code — https://github.com/Longxmas/SlimInfer

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. 2025a; 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. 2025b). However, these approaches often rely on predictor-based mechanisms, introducing additional overhead and complexity. Therefore, existing token pruning

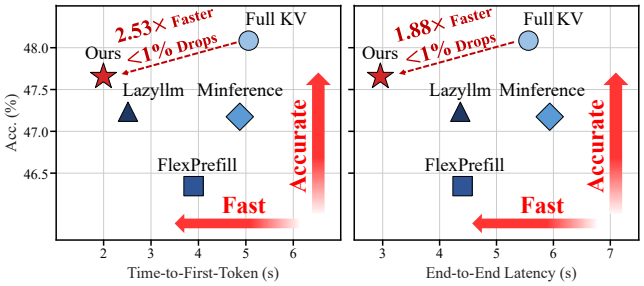

Figure 1: Accuracy *vs.* inference efficiency across different acceleration approaches on LongBench (Bai et al. 2024) for LLaMA-3.1-8B-Instruct (Grattafiori et al. 2024).

*Corresponding authors are Ao Zhou and Jianlei Yang.
Email: aozhou@buaa.edu.cn, jianlei@buaa.edu.cn.

methods still struggle to simultaneously optimize inference speed (Jiang et al. 2024; Huang et al. 2025c; Chen et al. 2025a,b), memory usage (Xiao et al. 2024b; Huang et al. 2025a), and model performance (Huang et al. 2025b; Wang et al. 2024).

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. 2025b) 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

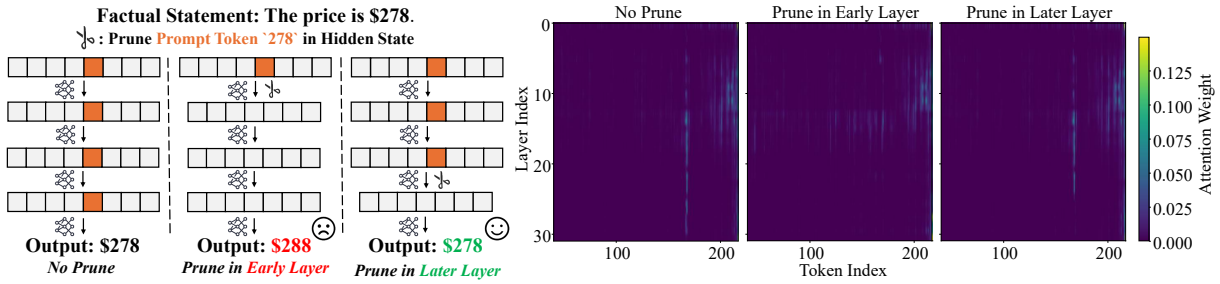

Figure 2: (**Left**) Illustration of a probing experiment on LLaMA3.1-8B-Instruct (Grattafiori et al. 2024). Pruning the hidden state of the critical prompt token "278" (which indicates the correct answer: "$278") in a later layer (right) results in the correct output, whereas pruning prompt tokens in 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.

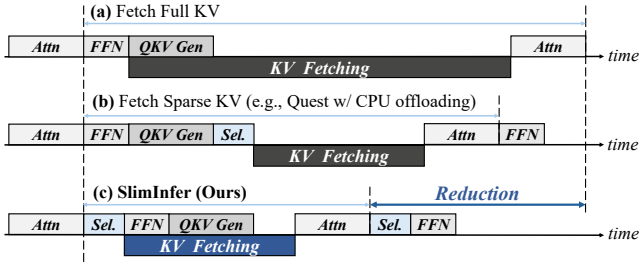

Figure 3: SlimInfer reduces the latency of a layer (*i.e.*, *QKV Generation+ Attention+FFN*) by prefetching KV cache that is offloaded to CPU, overlapping KV cache fetching with computation. "*Sel.*" means selecting tokens in KV cache (b) or hidden state (c) to prune.

decoding token towards the critical prompt token ("278" in the response → "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 *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. 2025b) 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

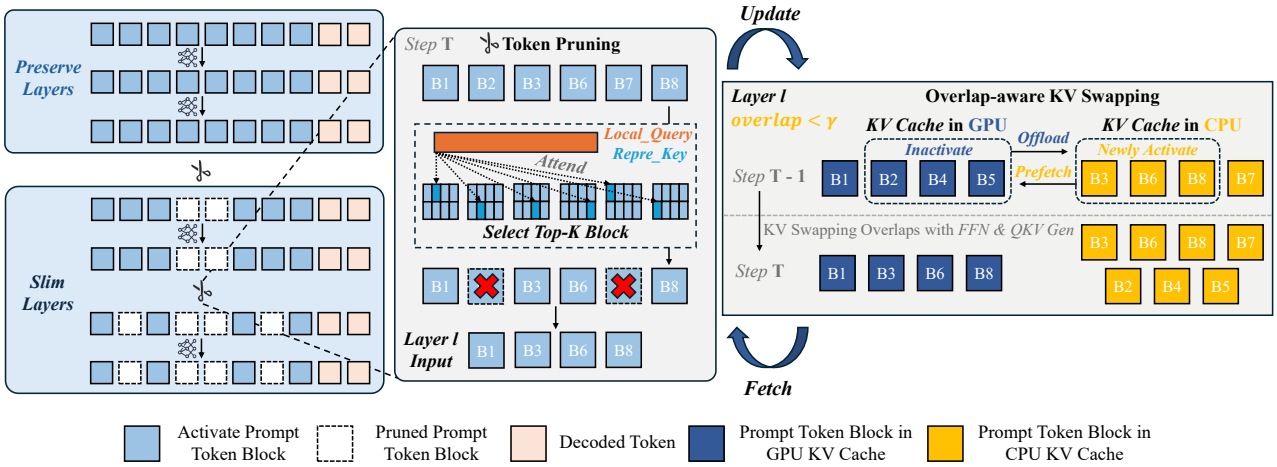

Figure 4: Overview of the proposed **SlimInfer**. (*i*) During inference, early *Preserve Layers* retain all prompt blocks to support *information diffusion* (Section 3.1), while later *Slim Layers* prune less relevant blocks to reduce computation (Section 4.1). (*ii*) Each block is divided into fine-grained *Token Units* for accurate importance scoring (Section 4.2). (*iii*) When `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.

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. 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. 2025b), 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.

[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

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 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

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.

surpassing other baselines on most benchmarks. These results underscore its broad generalization capability across different model architectures.

5.3 Efficiency Evaluation

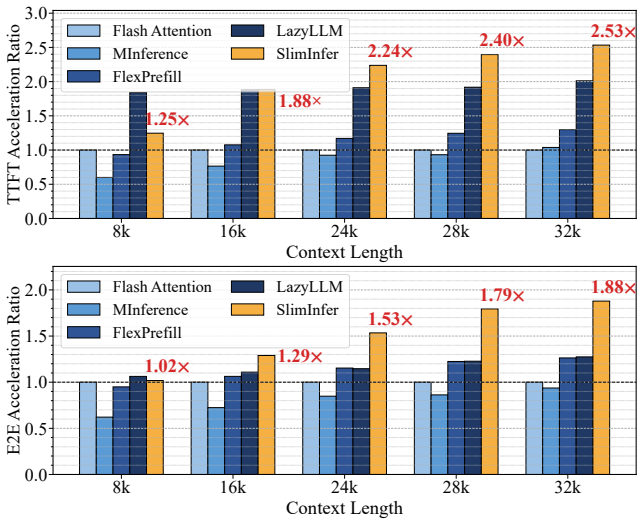

Figure 5: Inference efficiency comparison for LLaMA3.1-8B-Instruct (Grattafiori et al. 2024). (**Upper**) TTFT and (**Lower**) E2E latency acceleration ratio *vs.* context length. SlimInfer far outperforms the baselines at long context lengths ($\geq$24k) and remains on par with them in other cases.

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. To assess how latency scales with input length, we use 5 truncated versions of a 32k token sequence sampled from LongBench. 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 FlashAttention2 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 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 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.

Memory Efficiency In addition to latency, we evaluate SlimInfer's GPU memory footprint against other representative methods. FlexPrefill and MInference optimize computation but retain the full KV cache throughout all layers, resulting in no memory savings. LazyLLM applies dynamic pruning but overlooks KV cache offloading, missing an oppor-

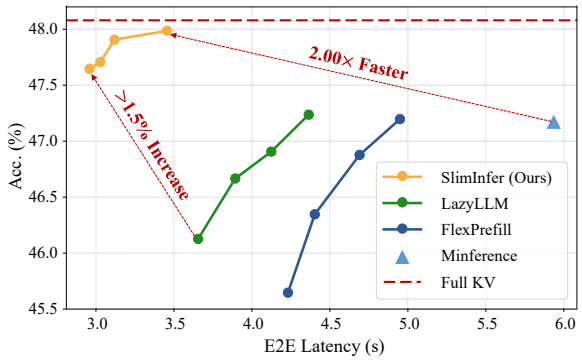

Figure 6: Accuracy *vs.* E2E latency for LLaMA3.1 on LongBench with 32k context. SlimInfer achieves a markedly superior trade-off, delivering near-lossless accuracy drops with substantially lower latency than other methods.

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.

tunity to reduce substantial GPU memory overhead. In contrast, SlimInfer combines a dynamic pruning strategy with offloading for KV pairs from inactive blocks to CPU memory. As shown in Table 2, this design yields 20.3−56.6% reductions in prompt KV cache memory.

5.4 Ablation Study

Balancing Pruning Depth and Token Retention We vary the pruning start layer while keeping the total number of retained tokens constant. This allows us to examine how the pruning position affects model performance across tasks. Specifically, pruning occurs only once at the start layer.

As shown in Figure 7, all three tasks exhibit a non-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 for LLaMA3.1-8B-Instruct.

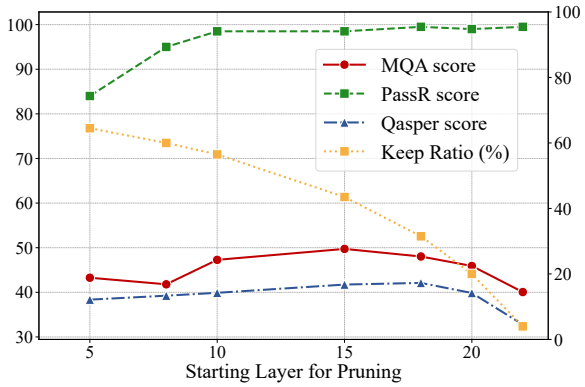

Figure 7: Impact of pruning start layer on LLaMA3.1-8B-Instruct task performance under fixed overall sparsity.

Block Importance Scoring Algorithm 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 inference latency with and without this optimization. As shown in Table 4, our method achieves consistent speedups over the FlashAttention baseline. At 32k context length, the speedup improves from 1.60× without asynchronous KV to 1.88× with it. These gains increase with input length, demonstrating the effectiveness of overlapping computation and data transfer.

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.

Acknowledgments

This work is supported in part by the National Key R&D Program of China (Grant No. 2023YFB4503704 and 2024YFB4505601), the National Natural Science Foundation of China (Grant No. 62572036), and the Beijing Natural Science Foundation (Grant No. L243031).

References

Bai, Y.; Lv, X.; Zhang, J.; Lyu, H.; Tang, J.; Huang, Z.; Du, Z.; Liu, X.; Zeng, A.; Hou, L.; et al. 2024. Longbench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 3119–3137.

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.

Chen, J.; Bai, S.; Wang, Z.; Wu, S.; Du, C.; Yang, H.; Gong, R.; Liu, S.; Wu, F.; and Chen, G. 2025a. Pre^3: Enabling Deterministic Pushdown Automata for Faster Structured LLM Generation. arXiv:2506.03887.

Chen, J.; Du, C.; Liu, R.; Yao, S.; Yan, D.; Liao, J.; Liu, S.; Wu, F.; and Chen, G. 2025b. TokenFlow: Responsive LLM Text Streaming Serving under Request Burst via Preemptive Scheduling. arXiv:2510.02758.

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.; Tao, D.; and Liu, X. 2024. Llmc: Benchmarking large language model quantization with a versatile compression toolkit. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, 132–152.

Grattafiori, A.; Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Vaughan, A.; et al. 2024. The llama 3 herd of models. *arXiv preprint 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. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 7362–7371.

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. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

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.; et al. 2024. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *Advances in Neural Information Processing Systems*, 37: 52481–52515.

Kryściński, W.; Rajani, N.; Agarwal, D.; Xiong, C.; and Radev, D. 2022. Booksum: A collection of datasets for long-form narrative summarization. In *Findings of the association for computational linguistics: EMNLP 2022*, 6536–6558.

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. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 155–172.

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. *Advances in Neural Information Processing Systems*, 37: 22947–22970.

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.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

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.

Wang, 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. In *Proceedings of the 32nd ACM International Conference on Multimedia*, 5742–5751.

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. *Advances in Neural Information Processing Systems*, 37: 119638–119661.

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. 2025a. TidalDecode: Fast and Accurate LLM Decoding with Position Persistent Sparse Attention. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

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

Yang, Z.; Qi, P.; Zhang, S.; Bengio, Y.; Cohen, W.; Salakhutdinov, R.; and Manning, C. D. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, 2369–2380.

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. *Proceedings of the ACM on Management of Data*, 3(3): 1–30.

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. 2020. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization. In *International conference on machine learning*, 11328–11339. PMLR.

Zhang, Z.; Sheng, Y.; Zhou, T.; Chen, T.; Zheng, L.; Cai, R.; Song, Z.; Tian, Y.; Ré, C.; Barrett, C.; et al. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36: 34661–34710.