HIADA: HISTORY-AWARE ADAPTIVE DECODING FOR DIFFUSION LARGE LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion large language models (dLLMs) enable parallel token generation through iterative denoising, but their practical inference remains costly due to dynamically changing bidirectional representations, the limited applicability of conventional KV caching, and rigid block-wise decoding schedules. These limitations call for inference strategies that can reuse computation while adapting to the evolving decoding state. We propose **HiAda**, a training-free, history-aware, and adaptive inference framework that exploits predictions and attention signals accumulated throughout the denoising trajectory. HiAda constructs a token-level active set to jointly coordinate KV-cache updates and token commitment, while dynamically adapting the decoding scope to the model's evolving confidence. This design reduces redundant computation, mitigates stale cache reuse, and improves the effective parallelism of dLLM decoding. Experiments on LLaDA-8B-Instruct and Dream-7B-Base across GSM8K, MATH, HumanEval, and MBPP show that HiAda achieves up to $51.1\times$ speedup over vanilla diffusion decoding and up to $2.01\times$ speedup over Fast-dLLM, while offering a controllable quality–efficiency trade-off. These results demonstrate that combining decoding history with adaptive computation can substantially improve the efficiency of diffusion language model inference.

1 INTRODUCTION

Large language models (LLMs) have achieved remarkable performance in natural language generation, code generation, and complex reasoning. Most modern LLMs rely on autoregressive (AR) generation, where tokens are generated sequentially under causal attention. While effective, AR decoding suffers from sequential dependencies and limited parallelism, diffusion large language models (dLLMs) have emerged as a promising alternative by formulating generation as an iterative denoising process.Unlike AR models, dLLMs can predict multiple tokens in parallel with bidirectional attention, offering a new trade-off between generation quality, global context modeling, and inference efficiency. Recent advances, including LLaDA (Nie et al., 2025b), Mercury (Khanna et al., 2025), and iLLaDA (Nie et al., 2026), demonstrate the scalability and competitiveness of dLLMs.

Despite their potential for parallel generation, dLLMs still suffer from substantial inference overhead. We identify three major limitations. First, bidirectional attention prevents directly applying conventional KV caching, causing repeated computation across denoising steps. Although recent works introduce approximate KV caching (Ma et al., 2025; Wu et al., 2025; Hu et al., 2026), their coarse-grained reuse strategies leave considerable redundancy. Second, fixed block-wise decoding confines computation to the current block, limiting parallelism near block completion while high-confidence positions beyond the boundary remain unused. Third, existing methods underutilize predictions at uncommitted positions, causing redundant recomputation across denoising steps. These challenges motivate a fine-grained decoding framework that can exploit both historical information and the inherent parallelism of dLLMs.

To address these issues, we propose HiAda, a history-aware adaptive decoding framework for dLLMs. HiAda leverages attention signals and historical predictions to make token-level decisions for token commitment and KV-cache updates. By removing fixed block constraints and adapting the decoding scope to the progress of decoding, HiAda better exploits parallel decoding opportunities while exposing an explicit and controllable trade-off between inference efficiency and generation

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

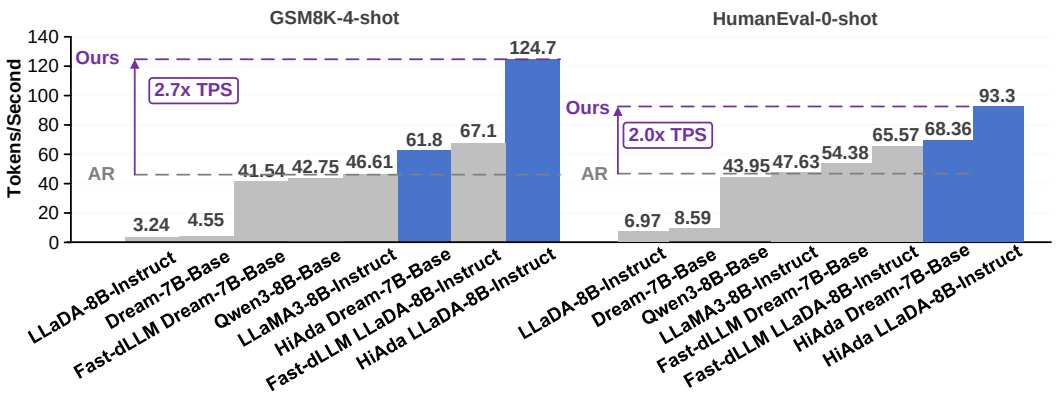


Figure 1: Throughput comparison on GSM8K (4-shot) and HumanEval (0-shot) with a generation length of 512 tokens. Gray bars represent the baseline methods, including vanilla dLLMs, Fast-dLLM, and autoregressive (AR) models, whereas blue bars highlight HiAda variants.HiAda achieves 124.7 and 93.3 tokens/s on GSM8K and HumanEval, respectively, yielding $2.7\times$ and $2.0\times$ speedups over the fastest AR baseline.

quality.As shown in Figure 1, HiAda reaches 124.7 and 93.3 tokens/s on GSM8K and HumanEval, respectively, surpassing the fastest evaluated AR baselines. Extensive experiments across reasoning and code-generation benchmarks further demonstrate substantial speedups over existing dLLM inference methods while offering a controllable quality–efficiency trade-off.

In summary, this work makes the following contributions:

- **Historical signals for efficient decoding.** We show that attention from recently decoded tokens predicts positions likely to be resolved next, while repeated token predictions provide a complementary signal of decoding reliability.
- **Unified token-level adaptive decoding.** We propose HiAda, a training-free framework that jointly coordinates masked-token prediction and KV-cache recomputation through token-level active sets, reducing redundant computation while controlling cache staleness.
- **Block-free parallel decoding.** HiAda replaces rigid block boundaries with an adaptive mask window and combines attention-guided position selection with history-aware confidence accumulation. This allows reliable tokens to be committed earlier and better exploits the parallelism of diffusion decoding.
- **End-to-end inference acceleration.** Across two dLLMs and four reasoning and code-generation benchmarks, HiAda achieves up to $51.1\times$ speedup over vanilla decoding and $2.01\times$ over Fast-dLLM. Together with the conservative HiAda* variant, the method provides a controllable quality–efficiency frontier.

2 RELATED WORK

Existing efforts to accelerate diffusion language model inference mainly fall into three categories: improving parallel decoding schedules, reusing or sparsifying intermediate computation, and introducing speculative or hybrid generation mechanisms.

Diffusion Large Language Models. Diffusion language models formulate text generation as an iterative denoising process over corrupted or partially masked sequences (Austin et al., 2021a; Sahoo et al., 2024). Masked diffusion models progressively replace `[MASK]` tokens using bidirectional sequence context, allowing multiple positions to be predicted in a single forward pass. Recent large-scale models, including LLaDA and Dream, demonstrate that this paradigm can achieve competitive performance on general language understanding, mathematical reasoning, and code generation (Nie et al., 2025b; Ye et al., 2025b). Nevertheless, realizing its theoretical parallelism in practice remains challenging because generation typically requires many repeated full-sequence evaluations.

Approximate KV Caching. Unlike autoregressive models, dLLMs cannot directly reuse KV states because of bidirectional attention. Existing methods therefore approximate KV reuse through selective cache refresh. Fast-dLLM freezes KV states outside the active decoding block (Wu et al., 2025), while dKV-Cache delays updates according to token representation dynamics (Ma et al., 2025). dLLM-Cache refreshes caches at adaptive intervals based on feature similarity (Liu et al., 2025), and more recent methods adopt fine-grained token- or layer-level refresh policies (Jiang et al., 2026; Nguyen-Tri et al., 2026). Despite these advances, existing methods often rely on predefined blocks, segments, or refresh intervals, which can lead to stale KV states or unnecessary recomputation. HiAda instead decouples masked-token decoding from KV recomputation at token-level granularity, enabling more precise and efficient cache updates.

Parallel Decoding. A common parallel decoding policy is confidence-based token commitment, where masked positions are selected according to prediction confidence.Fast-dLLM adopts threshold-based rules to determine how many high-confidence tokens are committed at each denoising step (Wu et al., 2025). Other methods further refine this decision using local confidence dynamics or historical predictions, such as AdaBlock-dLLM (Lu et al., 2026), CreditDecoding, and DecoCal (Wang et al., 2026; Xu et al., 2026). HiAda leverages historical predictions to identify more confident positions for commitment, sustaining higher parallelism.

3 MOTIVATION

3.1 PRELIMINARIES:INFERENCE PARADIGM OF DLLMS.

Unlike autoregressive LLMs, which generate one token at a time in a left-to-right order, dLLMs formulate generation as an iterative denoising process over a partially masked sequence. Let $\mathcal{V}$ denote the vocabulary and let $[\texttt{MASK}] \in \mathcal{V}$ be the special mask token. Given an input prompt $\mathbf{p}$ and a generation length budget L, inference starts from an initial sequence in which the prompt tokens are preserved and the target positions are fully masked:

$$\mathbf{x}^{(T)} = (\mathbf{p}, \underbrace{[\texttt{MASK}], \ldots, [\texttt{MASK}]}_{L \text{ positions}}), \tag{1}$$

where T denotes the total number of denoising steps.

The model then progressively transforms $\mathbf{x}^{(T)}$ into a fully denoised sequence $\mathbf{x}^{(0)}$ by repeatedly filling a subset of masked positions according to the current sequence context. At each denoising step t, the model performs a forward pass over $\mathbf{x}^{(t)}$ and produces a token distribution for every position. (Nie et al., 2025b; Ye et al., 2025b) For each masked position i, the model selects the most likely token and its associated confidence score:

$$\begin{aligned} \hat{x}_i^{(t-1)} &= \arg\max_{v \in \mathcal{V}} p_\theta(x_i = v \mid \mathbf{x}^{(t)}), \\ c_i^{(t-1)} &= \max_{v \in \mathcal{V}} p_\theta(x_i = v \mid \mathbf{x}^{(t)}). \end{aligned} \tag{2}$$

Here, $\hat{x}_i^{(t-1)}$ is the candidate token predicted for position i, and $c_i^{(t-1)}$ measures the corresponding confidence score. The next sequence state is obtained by an unmasking rule that decides whether a candidate prediction is sufficiently reliable to be committed. A common strategy, adopted by prior accelerated dLLM decoders such as Fast-dLLM (Wu et al., 2025), uses threshold-based sampling with a confidence threshold τ:

$$x_i^{(t-1)} = \begin{cases} \hat{x}_i^{(t-1)}, & x_i^{(t)} = [\texttt{MASK}] \text{ and } c_i^{(t-1)} \geq \tau, \\ [\texttt{MASK}], & x_i^{(t)} = [\texttt{MASK}] \text{ and } c_i^{(t-1)} < \tau, \\ x_i^{(t)}, & x_i^{(t)} \neq [\texttt{MASK}]. \end{cases} \tag{3}$$

3.2 OBSERVATIONS

During each denoising iteration, a dLLM forward pass serves two distinct but coupled computational purposes. First, it updates the key–value (KV) states induced by the bidirectional attention. Second,

it predicts and commits tokens at a subset of currently masked positions. A vanilla decoder performs both operations densely by recomputing all sequence positions at every iteration.

Prior studies on approximate KV caching have shown that KV states exhibit substantial temporal redundancy and that cache updates can be selectively scheduled according to representation dynamics or feature similarity (Ma et al., 2025; Liu et al., 2025; Jiang et al., 2026; Nguyen-Tri et al., 2026). Our analysis further reveals that, for many positions and decoding stages, the change in the KV representation between consecutive denoising steps is small, although the magnitude and spatial pattern of this change depend on the layer and decoding stage (see Appendix A.5). Therefore, recomputing every position at every iteration is unnecessarily expensive.

This analysis suggests separating the positions used for token prediction from those used for KV-cache recomputation. In particular, an efficient decoder should identify (i) masked positions that are most likely to be decoded in the next iteration and (ii) unmasked positions whose KV states are most likely to have changed substantially under the updated context. Only these positions need to be processed next, while the remaining KV states can be reused from the cache. HiAda is designed around this separation: its active mask set supports token prediction, whereas its active cache set refreshes the KV states of recently decoded positions.

Observation 1: Current attention is predictive of future decoding positions. Positions decoded at step t tend to assign relatively high attention to masked positions that are decoded in the subsequent step. We quantify the effectiveness of attention-guided position selection using the *hit ratio*. With parallel decoding disabled, the hit ratio is defined as the fraction of tokens produced by our restricted decoder that match the tokens produced by vanilla dLLM. As shown in Figure 3a–b, the hit ratio remains high across different active mask set budgets and attention-layer choices. Moreover, a single late Transformer layer provides a sufficiently strong attention signal, motivating the compact historical state used by HiAda.

Observation 2: Fixed-block boundaries limit decoding quality and parallelism. Under fixed-block decoding, token selection is restricted to the remaining masked positions within the current block. As denoising proceeds, this candidate set gradually shrinks, reducing decoding parallelism until the entire block is resolved and the decoder moves to the next one. Consequently, the decoder may be forced to commit low-confidence tokens near the end of a block, while more predictable positions beyond the block boundary remain unavailable. We therefore replace fixed blocks with an adaptive mask window containing the leftmost L_W remaining masked positions. The window is continuously replenished as tokens are committed, maintaining a stable candidate scope from which the next active positions are selected using historical attention.

4 METHODOLOGY

We propose HiAda, a history-aware adaptive inference framework for efficient dLLM decoding. HiAda uses a token-level active set composed of an attention-guided mask set for token prediction and a cache set that retains the most recently unmasked positions for KV recomputation. An adaptive mask window removes fixed block boundaries and evolves with decoding progress, while historical prediction consistency is accumulated to guide parallel unmasking. Together, these designs support sparse computation, fine-grained cache updates, and adaptive parallel decoding. The complete inference procedure of HiAda is provided in Appendix A.1.

4.1 ATTENTION-GUIDED ACTIVE SET

We define the *active set* as the set of token actually processed in a denoising step. A vanilla dLLM performs a full-sequence forward pass, and thus treats every position as active (Nie et al., 2025b; Ye et al., 2025b). With approximate KV caching, however, computation can be restricted to a subset of positions . For instance, Fast-dLLM activates the current decoding block and reuses cached states for positions outside the block (Wu et al., 2025). Recent methods further reduce the active set according to hidden-state variation or token confidence (Zhu et al., 2026; Long et al., 2026). These strategies nevertheless provide only partial sparsity: hidden-state-based skipping is applied to a subset of model layers and requires frequent cache refreshes, whereas confidence-guided context selection sparsifies attention but does not reduce computation in feed-forward network (FFN) layers.

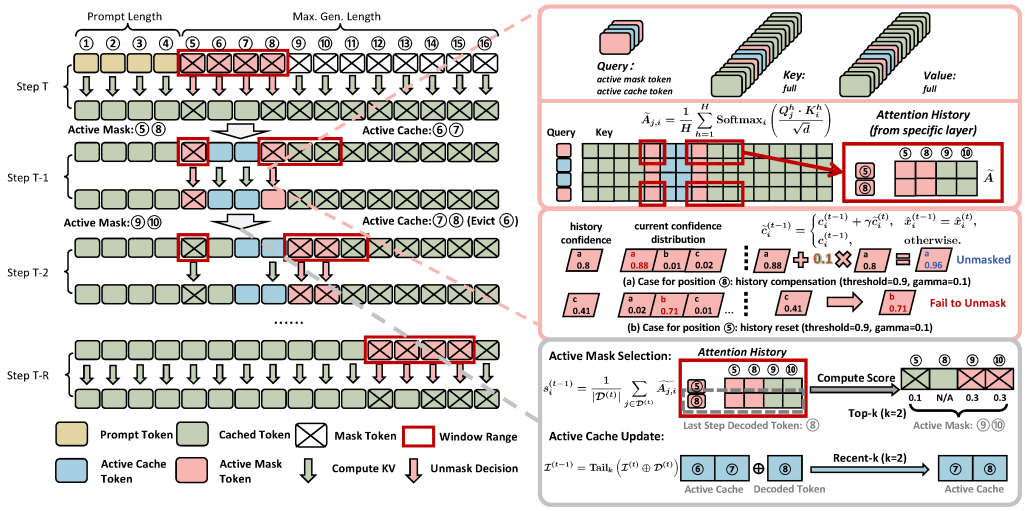


Figure 2: Overview of HiAda. HiAda uses attention history from a designated layer to update the active mask and cache sets. The example uses sets of size 2, an adaptive window of size 4, and a confidence threshold of 0.9, illustrating both full-sequence refresh and regular decoding.

To support end-to-end sparse computation, we decompose the active set at step t into two subsets:

$$\mathcal{A}^{(t)} = \mathcal{M}^{(t)} \cup \mathcal{I}^{(t)}. \tag{4}$$

The *active mask set* $\mathcal{M}^{(t)}$ contains masked positions selected for the current forward pass. The model evaluates their token distributions and confidence scores, after which the decoding rule determines which candidates are unmasked. The *active cache set* $\mathcal{I}^{(t)}$ is the KV-recomputation set, where positions have their KV states recomputed at the current denoising step. This allows the cache to incorporate up-to-date semantic information under the latest context. Since the former contains only masked positions and the latter only unmasked positions, the two sets are disjoint by construction.

We construct the active mask set using attention patterns observed at the preceding denoising step. Let $\mathcal{D}^{(t)}$ denote the positions unmasked from $\mathbf{x}^{(t)}$ to $\mathbf{x}^{(t-1)}$. At the designated attention layer ℓ^*, let $\mathbf{Q}_j^{h,(t)}$ and $\mathbf{K}_i^{h,(t)}$ denote the query at position j and the key at position i for head h, respectively. We define the *head-averaged attention weight* from query position j to key position i as

$$\widetilde{A}_{j,i}^{(t)} = \frac{1}{H} \sum_{h=1}^{H} \mathrm{Softmax}_i \left(\frac{\mathbf{Q}_j^{h,(t)} \cdot \mathbf{K}_i^{h,(t)}}{\sqrt{d}} \right) \tag{5}$$

For each position that remains masked, we measure its relevance to the newly decoded set by averaging the head-averaged attention weights over the decoded query positions:

$$s_i^{(t-1)} = \frac{1}{|\mathcal{D}^{(t)}|} \sum_{j \in \mathcal{D}^{(t)}} \widetilde{A}_{j,i}^{(t)}. \tag{6}$$

A high score indicates that the position is strongly attended to by positions resolved in the latest step and is therefore likely to be semantically relevant to the next stage of denoising. We construct the active mask set by selecting the k highest-scoring positions that remain masked:

$$\mathcal{M}^{(t-1)} = \underset{i:\, x_i^{(t-1)} = \texttt{[MASK]}}{\mathrm{TopK}} \left(s_i^{(t-1)}, k \right). \tag{7}$$

Here, k is the active-mask budget. The resulting top-k positions are evaluated in the forward pass, and the decoding policy selects the positions to unmask exclusively from this set.

We maintain the active cache set as a FIFO queue containing the k most recently unmasked positions:

$$\mathcal{I}^{(t-1)} = \mathrm{Tail}_k \left(\mathcal{I}^{(t)} \oplus \mathcal{D}^{(t)} \right), \tag{8}$$

where $\oplus$ denotes ordered concatenation. We allocate the same budget k to the active cache set $\mathcal{I}^{(t)}$ to ensure that the positions decoded in the current step, of which there are at most k, are retained for at least one KV-state update. The next denoising step consequently processes $\mathcal{A}^{(t-1)} = \mathcal{M}^{(t-1)} \cup \mathcal{I}^{(t-1)}$, jointly supporting token prediction at highly promising masked positions and KV-cache update at recently decoded positions.

4.2 Adaptive Mask Window

To overcome the fixed-block decoding limitations discussed in Observation 3.2, we replace static block boundaries with an *adaptive mask window* that evolves with decoding progress. Let

$$\mathcal{U}^{(t)} = \left\{ i \mid x_i^{(t)} = [\texttt{MASK}] \right\} = \left\{ i_1^{(t)}, \ldots, i_{N_t}^{(t)} \right\}, \quad i_1^{(t)} < \cdots < i_{N_t}^{(t)}, \tag{9}$$

denote the ordered set of masked positions at step t. Given a window size $L_{\mathcal{W}}$, we define

$$\mathcal{W}^{(t)} = \left\{ i_r^{(t)} \;\middle|\; 1 \le r \le \min\left(L_{\mathcal{W}}, N_t\right) \right\}. \tag{10}$$

Thus, $\mathcal{W}^{(t)}$ contains the leftmost $L_{\mathcal{W}}$ masked positions, or all remaining masked positions when fewer than $L_{\mathcal{W}}$ remain. Unlike a static block, the window advances without waiting for all positions currently inside it to be decoded. After each denoising step, decoded positions are removed and replaced by the earliest masked positions beyond the current window. Hence, $|\mathcal{W}^{(t)}| = L_{\mathcal{W}}$ whenever sufficiently many masked positions remain.

The attention-guided active mask set is then selected within this window:

$$\mathcal{M}^{(t-1)} = \underset{i \in \mathcal{W}^{(t-1)}}{\mathrm{TopK}} \left(s_i^{(t-1)}, \min\left(k, \left|\mathcal{W}^{(t-1)}\right|\right) \right). \tag{11}$$

This windowed selection removes the parallel-decoding constraint imposed by fixed block boundaries while limiting attention-score maintenance to a bounded candidate region. In particular, the required attention-score storage is reduced from $\mathcal{O}(N^2)$ for global retention to $\mathcal{O}(NL_{\mathcal{W}})$, where N is the maximum generation length. Since $L_{\mathcal{W}} \ll N$ in practice, the resulting memory overhead is substantially smaller and does not become a bottleneck. A detailed derivation is provided in Appendix A.2.

4.3 History-aware Parallel Decoding

We extend the standard threshold-based parallel decoding rule by incorporating historical prediction consistency into the confidence score. Instead of deciding token commitment solely from the current confidence, we accumulate evidence when a masked position repeatedly predicts the same token.

For an active masked position $i \in \mathcal{M}^{(t)}$, let $\hat{x}_i^{(t-1)}$ and $c_i^{(t-1)}$ denote the current candidate token and its raw confidence. We define the history-aware confidence as

$$\widetilde{c}_i^{(t-1)} = \begin{cases} c_i^{(t-1)} + \gamma \widetilde{c}_i^{(t)}, & \hat{x}_i^{(t-1)} = \hat{x}_i^{(t)}, \\ c_i^{(t-1)}, & \text{otherwise}, \end{cases} \tag{12}$$

where $\gamma \in [0, 1)$ controls the contribution of historical evidence. Here, $\hat{x}_i^{(t)}$ and $\widetilde{c}_i^{(t)}$ are the candidate and accumulated score stored from the most recent evaluation of position i. If the predicted token changes, the accumulated history is reset.

We then apply the original threshold-based commitment rule to the accumulated score:

$$\mathcal{D}^{(t)} = \left\{ i \in \mathcal{M}^{(t)} \;\middle|\; \widetilde{c}_i^{(t-1)} \ge \tau \right\} \cup \left\{ \underset{i \in \mathcal{M}^{(t)}}{\arg\max}\, \widetilde{c}_i^{(t-1)} \right\}. \tag{13}$$

Each $i \in \mathcal{D}^{(t)}$ is assigned $x_i^{(t-1)} = \hat{x}_i^{(t-1)}$. Thus, the threshold τ remains unchanged, while predictions that stay consistent across evaluations are selectively promoted. Note that $\widetilde{c}_i^{(t-1)}$ is a decision score rather than a calibrated probability and may exceed one.

4.4 Periodic Full KV Refresh

Approximate KV caching inevitably introduces stale representations. Existing methods therefore periodically perform full-sequence forward passes to refresh the KV cache (Wu et al., 2025). Our analysis further shows that KV states continue to change throughout different stages of decoding (see A.5), indicating that full refresh remains necessary even with fine-grained token-level updates.Accordingly, HiAda defines its full-refresh interval as a function of k, balancing sparse updates with periodic global recomputation. We further introduce an accuracy-oriented variant, **HiAda***, which adopts a more conservative refresh strategy to better preserve generation quality.

5 Experiment

We conduct all experiments on an NVIDIA A6000 GPU and evaluate HiAda on two representative dLLMs, LLaDA-8B-Instruct (Nie et al., 2025a) and Dream-7B-Base (Ye et al., 2025a). We consider four widely used benchmarks covering mathematical reasoning and code generation, including GSM8K (Cobbe et al., 2021), MATH (Hendrycks et al., 2021), HumanEval (Chen et al., 2021), and MBPP (Austin et al., 2021b). All evaluations are performed using the lm-evaluation-harness framework (Gao et al., 2024) to ensure standardized and reproducible evaluation.

We compare HiAda with the vanilla implementations of LLaDA and Dream, as well as Fast-dLLM (Wu et al., 2025), a representative training-free dLLM acceleration method combining approximate KV caching and parallel decoding. We additionally evaluate HiAda against two more recent training-free acceleration methods, Dynamic-dLLM (Wu et al., 2026) and ES-dLLM (Zhu et al., 2026). To keep the main comparison focused, their end-to-end results and complete evaluation configurations are provided in Appendix A.6.

5.1 Main results

Table 1 reports the main results on LLaDA-8B-Instruct and Dream-7B-Base. Following prior work, we report throughput in tokens per second (TPS), speedup over the corresponding vanilla implementation under the same benchmark and generation length, and the corresponding task-performance score. HiAda denotes our efficiency-oriented configuration with a fixed refresh schedule, whereas HiAda* denotes the accuracy-oriented strategy. The adaptive schedule allocates additional full-cache refreshes to later denoising steps, where accurate state updates become increasingly important, providing a more conservative accuracy–efficiency trade-off. The complete experimental hyperparameters are reported in Appendix A.4. Additional comparisons with Dynamic-dLLM and ES-dLLM under aligned hardware and decoding settings are presented in Appendix A.6.

Across the evaluated settings, HiAda provides the highest throughput, while HiAda* generally recovers part of the performance loss at a moderate throughput cost. For example, on LLaDA-8B-Instruct with MBPP and 512 generated tokens, HiAda reaches $51.1\times$ speedup with a 3.7-point performance decrease, whereas HiAda* achieves $43.0\times$ speedup with a smaller 2.0-point decrease. This illustrates that the main benefit of HiAda is a controllable operating frontier rather than a single universally optimal configuration.

5.2 Ablation and Analysis

Component-wise ablation. We conduct an incremental ablation study on Dream-7B-Base to isolate the contribution of each component of HiAda. The results are reported in Table 2. We first include vanilla diffusion and DualCache as reference configurations. We then replace the fixed-block DualCache computation with the attention-guided active set and progressively introduce standard parallel decoding, history-aware confidence accumulation, and the adaptive mask window. Therefore, beginning with the attention-guided active-set configuration, each subsequent row contains all components from the preceding row.

Effect of k and history layer on the hit ratio. We define the *hit ratio* as the fraction of positions selected by the predicted active mask set that are decoded in the subsequent denoising step. As shown in Figure 3a–b, the hit ratio generally improves with k and gradually saturates, while remaining high across a range of history layers. These results indicate sensitivity to the active-set budget but relative

LLaDA-8B-Instruct

Benchmark	Len.	Method	TPS	Speedup	Score
GSM8K	256	LLaDA	3.85	1.0 ×	80.5
		Fast-dLLM	48.5	12.6 ×	**80.7**
		HiAda	**67.7**	**17.6** ×	80.4
		HiAda*	49.2	12.8 ×	**80.7**
	512	LLaDA	3.24	1.0 ×	**82.3**
		Fast-dLLM	67.1	20.7 ×	80.8
		HiAda	**124.7**	**38.5** ×	79
		HiAda*	99.7	30.8 ×	81.4
MATH	256	LLaDA	6.92	1.0 ×	**38.1**
		Fast-dLLM	141.5	20.4 ×	34.7
		HiAda	**226.5**	**32.7** ×	36.2
	512	LLaDA	5.41	1.0 ×	**36.4**
		Fast-dLLM	109.6	20.3 ×	34.7
		HiAda	**210.2**	**38.9** ×	34.4
		HiAda*	144.9	26.8 ×	34.9
HumanEval	256	LLaDA	11.15	1.0 ×	**44.5**
		Fast-dLLM	53.6	4.8 ×	39
		HiAda	**69.7**	**6.3** ×	40.9
	512	LLaDA	6.97	1.0 ×	**42.7**
		Fast-dLLM	65.57	9.4 ×	39
		HiAda	**93.3**	**13.4** ×	**42.7**
MBPP	256	LLaDA	4.8	1.0 ×	**39**
		Fast-dLLM	82	17.1 ×	34.8
		HiAda	**130**	**27.1** ×	37.4
	512	LLaDA	3.9	1.0 ×	**38.6**
		Fast-dLLM	99	25.4 ×	36.2
		HiAda	**199.1**	**51.1** ×	35.9
		HiAda*	167.7	43.0 ×	36.6

Dream-7B-Base

Benchmark	Len.	Method	TPS	Speedup	Score
GSM8K	256	Dream	5.63	1.0 ×	70.36
		Fast-dLLM	41.43	7.36 ×	72.18
		HiAda	**52.38**	**9.3** ×	**72.71**
	512	Dream	4.55	1.0 ×	75.44
		Fast-dLLM	41.54	9.13 ×	76.57
		HiAda	**61.80**	**13.58** ×	**76.65**
MATH	256	Dream	6.2	1.0 ×	**39.62**
		Fast-dLLM	54.82	8.84 ×	38.6
		HiAda	**77.29**	**12.47** ×	38.16
		HiAda*	54.73	8.83 ×	38.44
	512	Dream	4.95	1.0 ×	**41.14**
		Fast-dLLM	64.59	13.05 ×	39.24
		HiAda	**108.79**	**21.98** ×	38.38
		HiAda*	89.56	18.1 ×	39.28
HumanEval	256	Dream	12.23	1.0 ×	51.83
		Fast-dLLM	54.10	4.42 ×	54.27
		HiAda	**72.37**	**5.92** ×	**55.49**
	512	Dream	8.59	1.0 ×	50.61
		Fast-dLLM	54.38	6.33 ×	53.05
		HiAda	**68.36**	**7.96** ×	**54.88**
MBPP	256	Dream	6.19	1.0 ×	51.8
		Fast-dLLM	56.62	9.14 ×	53.6
		HiAda	**74.59**	**12.05** ×	**55**
	512	Dream	4.95	1.0 ×	53
		Fast-dLLM	65.22	13.18 ×	52
		HiAda	**112.40**	**22.71** ×	**53.6**

Table 1: End-to-end inference performance of HiAda and HiAda* on LLaDA-8B-Instruct and Dream-7B-Base across mathematical-reasoning and code-generation benchmarks. We report throughput in tokens per second (TPS), speedup relative to the corresponding vanilla diffusion baseline, and the task-performance score. For each model, benchmark, and maximum generation-length setting, the highest throughput and the best task-performance score are highlighted in bold.

Table 2: Component-wise ablation of HiAda on Dream-7B-Base. Each cell reports task score and throughput in tokens per second, with the relative speedup over Vanilla diffusion.

Dataset	Vanilla diffusion	DualCache	Attention-guided active set	+ Parallel decoding	+ History-aware confidence	+ Adaptive mask window
MBPP	51.80	52.40	**55.40**	**55.40**	55.00	55.00
	6.19 (1.00×)	21.88 (3.54×)	31.30 (5.06×)	66.53 (10.75×)	74.36 (12.01×)	**76.32** (12.33×)
GSM8K	70.36	71.40	70.60	72.70	71.60	**72.71**
	5.63 (1.00×)	28.07 (4.99×)	27.67 (4.92×)	46.40 (8.24×)	49.06 (8.72×)	**52.38** (9.30×)

robustness to the selected attention layer, supporting our single-layer design. Additional results are provided in Appendix A.3.

Effect of k on end-to-end performance. As shown in Figure 3c–d, a moderate k improves decoding parallelism and cache freshness, yielding higher throughput while generally preserving task performance. When k becomes too large, the additional computation and refresh overhead reduce throughput with only limited performance gains. Thus, an intermediate k provides the most favorable efficiency–quality trade-off.

Effect of γ on end-to-end performance. As shown in Figure 3e–f, increasing γ generally improves throughput by promoting historically consistent predictions. Task performance is non-monotonic, however, because an overly large γ may overemphasize stale historical predictions. A moderate value therefore provides a favorable efficiency–quality trade-off.

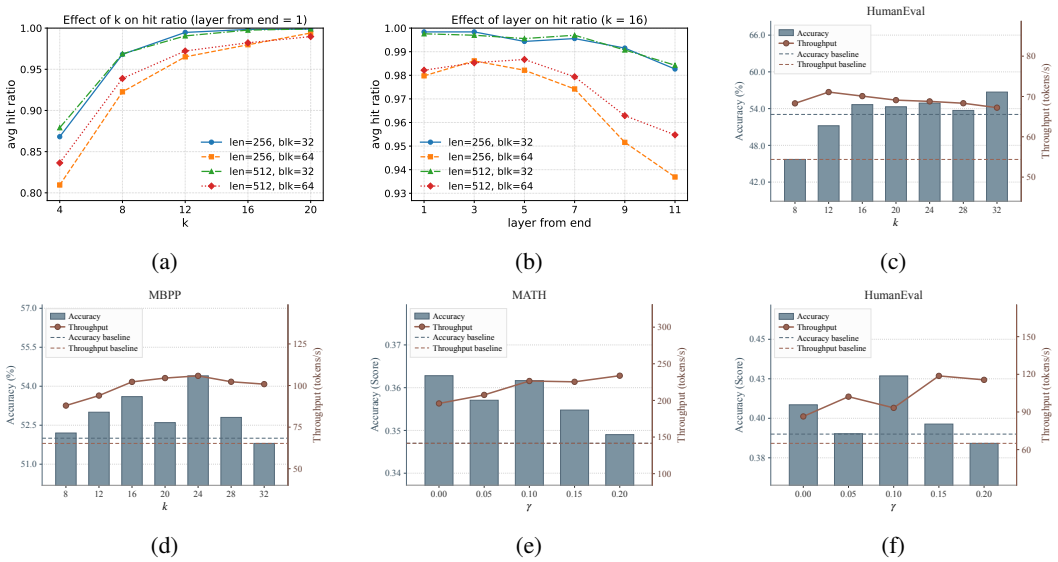


Figure 3: Ablation of the attention-guided position predictor and key HiAda hyperparameters. **(a)–(b)** Hit ratio under different active-set budgets k and Transformer layers, evaluated with vanilla LLaDA-8B-Instruct. **(c)–(d)** Effect of k on HumanEval and MBPP using Dream-7B-Base. **(e)–(f)** Effect of the history factor γ on MATH and HumanEval using LLaDA-8B-Instruct. For **(c)–(f)**, bars and curves show task performance and throughput, respectively; dashed lines indicate the Fast-dLLM throughput.

Table 3: Summary of the memory, cumulative query-token evaluations, and FLOP complexity.

Metric	Vanilla	Fast-dLLM	HiAda
Regular attention memory	$\mathcal{O}(N^2)$	$\mathcal{O}(BN)$	$\mathcal{O}(kN)$
Extra state beyond KV cache	–	–	$\mathcal{O}(k^2 + kL_w + N)$
Cumulative query-token evaluations	$\mathcal{O}(N^2)$	$\mathcal{O}(\frac{BN}{n_{\mathrm{F}}} + \frac{N^2}{B})$	$\mathcal{O}(\frac{kN}{n_{\mathrm{H}}} + F_{\mathrm{H}}N)$
Cumulative attention FLOPs	$\mathcal{O}(N^3 d)$	$\mathcal{O}(\frac{BN^2 d}{n_{\mathrm{F}}} + \frac{N^3 d}{B})$	$\mathcal{O}(\frac{kN^2 d}{n_{\mathrm{H}}} + F_{\mathrm{H}}N^2 d)$

6 Discussion

Efficiency and resource trade-offs. Table 3 separates three costs that can otherwise be conflated: regular-step attention memory, token-wise dense computation, and attention FLOPs. The central result is conditional: when the denoising horizon is proportional to N, k,n_{F} and n_{H} remain fixed, and the fixed-refresh schedule yields $F_{\mathrm{H}} \approx k/n_{\mathrm{H}}$, HiAda changes the sequence-length dependence of cumulative token-wise dense computation from quadratic to linear and that of cumulative attention from cubic to quadratic. every active query still attends to all N cached key–value positions. The additional history state is lower order under $k, L_w \ll N$, but the asymptotic analysis alone does not capture indexing,cache-management, or hardware-dependent kernel overheads. Detailed explanation can be found in A.2

Limitations and future directions. HiAda still requires periodic full-sequence refreshes because representations evolve throughout denoising; consequently, its $\mathcal{O}(kN)$ regular-step attention memory does not remove the $\mathcal{O}(N^2)$ peak-memory cost of a refresh. Its speedup also comes with a task-dependent quality trade-off: aggressive sparsity and reuse can amplify stale-state or erroneous-history effects, while conservative refreshes recover quality at additional cost. Moreover, the scaling simplification above treats n_{H} as fixed, an assumption that may not hold across sequence lengths, models, or data distributions.

AI USE STATEMENT

Generative AI tools were used to assist with language editing, manuscript organization, LaTeX formatting, and the development and debugging of selected parts of the experimental code. All AI-assisted code was manually reviewed, tested, and validated by the authors before being used in the experiments. The proposed method, experimental design, execution, result analysis, and scientific conclusions were determined and verified by the authors. Generative AI tools were not used to generate, alter, or fabricate experimental measurements. All AI-assisted manuscript content was also reviewed and revised for technical correctness, appropriate attribution, and consistency with the reported experiments. The authors take full responsibility for the final content of this work, including all text, code, results, and claims produced with the assistance of generative AI.

REPRODUCIBILITY STATEMENT

We provide the complete HiAda inference procedure and implementation details in Section 4 and Appendix A.1. The storage and computational-complexity analyses, including their assumptions and derivations, are presented in Appendix A.2. Detailed hyperparameters for LLaDA-8B-Instruct, Dream-7B-Base, Fast-dLLM, HiAda, and HiAda* are reported in Appendix A.4. The configurations used for the additional comparisons with Dynamic-dLLM and ES-dLLM are provided in Appendix A.6. We report the evaluated model checkpoints, benchmarks, generation lengths, denoising steps, decoding thresholds, cache and window settings, batch size, and hardware environment. All experiments are conducted using standardized benchmark protocols based on the LM Evaluation Harness.An anonymized implementation, evaluation scripts, and configuration files are included in the supplementary material.

REFERENCES

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. In *Advances in Neural Information Processing Systems*, volume 34, pp. 17981–17993, 2021a.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021b.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024. URL `https://zenodo.org/records/12608602`.

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

Zhanqiu Hu, Jian Meng, Yash Akhauri, Mohamed S. Abdelfattah, Jae-sun Seo, Zhiru Zhang, and Udit Gupta. Flashdlm: Accelerating diffusion language model inference via efficient kv caching and guided diffusion. In *International Conference on Learning Representations*, 2026.

Yuchu Jiang, Yue Cai, Xiangzhong Luo, Jiale Fu, Jiarui Wang, Chonghan Liu, and Xu Yang. d^2cache: Accelerating diffusion-based llms via dual adaptive caching. In *International Conference on Learning Representations*, 2026.

Samar Khanna, Siddhant Kharbanda, Shufan Li, Harshit Varma, Eric Wang, Sawyer Birnbaum, Ziyang Luo, Yanis Miraoui, Akash Palrecha, Stefano Ermon, Aditya Grover, and Volodymyr Kuleshov. Mercury: Ultra-fast language models based on diffusion. *arXiv preprint arXiv:2506.17298*, 2025.

Zhiyuan Liu, Yicun Yang, Yaojie Zhang, Junjie Chen, Chang Zou, Qingyuan Wei, Shaobo Wang, and Linfeng Zhang. dllm-cache: Accelerating diffusion large language models with adaptive caching. *arXiv preprint arXiv:2506.06295*, 2025.

Lingkun Long, Yushi Huang, Shihao Bai, Ruihao Gong, Jun Zhang, Ao Zhou, and Jianlei Yang. Focus-dllm: Accelerating long-context diffusion llm inference via confidence-guided context focusing, 2026. URL https://arxiv.org/abs/2602.02159.

Guanxi Lu, Hao Mark Chen, Yuto Karashima, Zhican Wang, Daichi Fujiki, and Hongxiang Fan. Adablock-dllm: Semantic-aware diffusion llm inference via adaptive block size. In *International Conference on Learning Representations*, 2026.

Xinyin Ma, Runpeng Yu, Gongfan Fang, and Xinchao Wang. dkv-cache: The cache for diffusion language models. In *Advances in Neural Information Processing Systems*, 2025.

Quan Nguyen-Tri, Mukul Ranjan, and Zhiqiang Shen. Attention is all you need for kv cache in diffusion llms. In *International Conference on Learning Representations*, 2026.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models, 2025a. URL https://arxiv.org/abs/2502.09992.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. *arXiv preprint arXiv:2502.09992*, 2025b.

Shen Nie, Qiyang Min, Shaoxuan Xu, Zihao Huang, Yuxuan Song, Yong Shan, Yankai Lin, Wayne Xin Zhao, Chongxuan Li, and Ji-Rong Wen. Improved large language diffusion models. *arXiv preprint arXiv:2606.25331*, 2026.

Subham Sekhar Sahoo, Marianne Arriola, Aaron Gokaslan, Edgar Marroquin, Justin T. Chiu, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. In *Advances in Neural Information Processing Systems*, 2024.

Kangyu Wang, Zhiyun Jiang, Haibo Feng, Weijia Zhao, Lin Liu, Jianguo Li, Zhenzhong Lan, and Weiyao Lin. Creditdecoding: Accelerating parallel decoding in diffusion large language models with trace credits. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, pp. 11105–11123, 2026.

Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. Fast-dllm: Training-free acceleration of diffusion llm by enabling kv cache and parallel decoding. *arXiv preprint arXiv:2505.22618*, 2025.

Tianyi Wu, Xiaoxi Sun, Yanhua Jiao, Yulin Li, Yixin Chen, Yunhao Cao, Yiqi Hu, and Zhuotao Tian. Dynamic-dllm: Dynamic cache-budget and adaptive parallel decoding for training-free acceleration of diffusion llm. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=SdnkB5pGbq.

Fan Xu, Huixuan Zhang, and Xiaojun Wan. Decocal: Decoding with calibration in diffusion large language models. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, pp. 11866–11880. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.545.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7b: Diffusion large language models, 2025a. URL https://arxiv.org/abs/2508.15487.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7b: Diffusion large language models. *arXiv preprint arXiv:2508.15487*, 2025b.

Zijian Zhu, Fei Ren, Zhanhong Tan, and Kaisheng Ma. Es-dllm: Efficient inference for diffusion large language models by early-skipping, 2026. URL https://arxiv.org/abs/2603.10088.

A APPENDIX

A.1 ALGORITHM DETAILS

Algorithm 1 summarizes the complete inference procedure of HiAda. The algorithm alternates between periodic full-sequence refreshes and sparse forward passes. During sparse inference, the active mask set is predicted from historical attention, while recently decoded positions in the active cache set are selected for KV-cache recomputation. Token commitment is further determined by the history-aware confidence score introduced in Section 4.3

Algorithm 1 HiAda with Periodic Full KV-Cache Refresh

Require: Prompt $\mathbf{p}$; generation length N; denoising steps T; active mask budget k; mask-window size $L_{\mathcal{W}}$; designated attention layer $\ell^\star$; confidence threshold τ; history factor γ, refresh strategy
Ensure: Fully denoised sequence $\mathbf{x}^{(0)}$

$\mathcal{G} \leftarrow \{|\mathbf{p}|+1, \ldots, |\mathbf{p}|+N\}$
$\mathbf{x}^{(T)} \leftarrow (\mathbf{p}, [\texttt{MASK}]^L)$
$\mathcal{I}^{(T)} \leftarrow \varnothing$
$(\hat{\mathbf{x}}^{(T)}, \widetilde{\mathbf{c}}^{(T)}) \leftarrow ([\varnothing]^L, \mathbf{0}_N)$
for $t = T, T-1, \ldots, 1$ **do**
  $\mathcal{U}^{(t)} \leftarrow \{i \in \mathcal{G} \mid x_i^{(t)} = [\texttt{MASK}]\}$
  $\mathcal{W}^{(t)} \leftarrow \textsc{LeftmostWindow}\left(\mathcal{U}^{(t)}, L_{\mathcal{W}}\right)$
  $\mathcal{W}_{\text{ext}}^{(t)} \leftarrow \textsc{LeftmostWindow}\left(\mathcal{U}^{(t)}, L_{\mathcal{W}}+k\right)$ ▷ Cover the next window under at most k commits
  $\mathbf{x}^{(t-1)} \leftarrow \mathbf{x}^{(t)}$
  $\hat{\mathbf{x}}^{(t-1)} \leftarrow \hat{\mathbf{x}}^{(t)}$
  $R \leftarrow \max\left(1, \left\lfloor \frac{|\mathcal{U}^{(t)}| \text{ if } \texttt{refresh_strategy=accuracy_oriented} \text{ else } N}{k} \right\rfloor\right)$
  if $(T-t) \bmod R = 0$ **then** // Refresh Step: Full Forward, Standard Parallel Decoding
    $(\hat{\mathbf{x}}_{\mathcal{W}^{(t)}}^{(t-1)}, \mathbf{c}_{\mathcal{W}^{(t)}}^{(t-1)}, \widetilde{\mathbf{A}}_{\mathcal{W}^{(t)}, \mathcal{W}_{\text{ext}}^{(t)}}^{(t)}, \mathcal{C}^{(t)}) \leftarrow \textsc{FullForward}\left(\mathbf{x}^{(t)}; \mathcal{W}^{(t)}, \mathcal{W}_{\text{ext}}^{(t)}\right)$
    $\mathcal{E}^{(t)} \leftarrow \underset{i \in \mathcal{W}^{(t)}}{\text{TopK}}\left(c_i^{(t-1)}, \min\left(k, |\mathcal{W}^{(t)}|\right)\right)$ ▷ Retain the top-k for unmasking
    $\mathcal{D}^{(t)} \leftarrow \left\{i \in \mathcal{E}^{(t)} \mid c_i^{(t-1)} \geq \tau\right\} \cup \left\{\underset{i \in \mathcal{E}^{(t)}}{\arg\max}\, c_i^{(t-1)}\right\}$
    $\widetilde{\mathbf{c}}^{(t-1)} \leftarrow \mathbf{0}, \quad \widetilde{\mathbf{c}}_{\mathcal{W}^{(t)}}^{(t-1)} \leftarrow \mathbf{c}_{\mathcal{W}^{(t)}}^{(t-1)}$ ▷ Reset confidence history
    $\mathcal{I}^{(t-1)} \leftarrow \mathcal{D}^{(t)}$ ▷ Reset active cache set
    $\mathbf{x}_{\mathcal{D}^{(t)}}^{(t-1)} \leftarrow \hat{\mathbf{x}}_{\mathcal{D}^{(t)}}^{(t-1)}$
  else // Sparse Step: Selective Forward, History-aware Parallel Decoding
    Compute $s_i^{(t)}$ for all $i \in \mathcal{W}^{(t)}$ from $\widetilde{\mathbf{A}}_{\mathcal{D}^{(t+1)}, \mathcal{W}^{(t)}}^{(t+1)}$ according to equation 6
    $\mathcal{M}^{(t)} \leftarrow \underset{i \in \mathcal{W}^{(t)}}{\text{TopK}}\left(s_i^{(t)}, \min\left(k, |\mathcal{W}^{(t)}|\right)\right)$

$\mathcal{A}^{(t)} \leftarrow \mathcal{M}^{(t)} \cup \mathcal{I}^{(t)}$

$(\hat{\mathbf{x}}^{(t-1)}_{\mathcal{M}^{(t)}}, \mathbf{c}^{(t-1)}_{\mathcal{M}^{(t)}}, \widetilde{\mathbf{A}}^{(t)}_{\mathcal{M}^{(t)},\mathcal{W}^{(t)}_{\text{ext}}}, \mathcal{C}^{(t)}_{\mathcal{A}^{(t)}}) \leftarrow \textsc{SparseForward}\left(\mathbf{x}^{(t)}, \mathcal{A}^{(t)}, \mathcal{C}^{(t+1)}; \mathcal{M}^{(t)}, \mathcal{W}^{(t)}_{\text{ext}}\right)$

Form $\mathcal{C}^{(t)}$ by replacing $\mathcal{C}^{(t+1)}_{\mathcal{A}^{(t)}}$ in $\mathcal{C}^{(t+1)}$ with $\mathcal{C}^{(t)}_{\mathcal{A}^{(t)}}$

Update $\widetilde{\mathbf{c}}^{(t-1)}$ on $\mathcal{M}^{(t)}$ by equation 12, carrying all other entries forward

$\mathcal{D}^{(t)} \leftarrow \left\{ i \in \mathcal{M}^{(t)} \mid \widetilde{c}^{(t-1)}_i \geq \tau \right\} \cup \left\{ \arg\max_{i \in \mathcal{M}^{(t)}} \widetilde{c}^{(t-1)}_i \right\}$

$\mathcal{I}^{(t-1)} \leftarrow \text{Tail}_k\left(\mathcal{I}^{(t)} \oplus \mathcal{D}^{(t)}\right)$ ▷ Update active cache set

$\mathbf{x}^{(t-1)}_{\mathcal{D}^{(t)}} \leftarrow \hat{\mathbf{x}}^{(t-1)}_{\mathcal{D}^{(t)}}$

end if

end for

return $\mathbf{x}^{(0)}$

Notation. All position sets use global sequence indices. LEFTMOSTWINDOW returns the requested number of leftmost masked positions; TopK returns the indices with the largest scores; $\oplus$ denotes ordered concatenation; and Tail_k keeps the last k indices. Here, $\mathcal{E}^{(t)}$ is the refresh-step candidate set, $\mathcal{M}^{(t)}$ is the active mask set, $\mathcal{I}^{(t)}$ is the active cache set, and $\mathcal{D}^{(t)}$ is the set committed at step t. $\hat{\mathbf{x}}^{(t)}$ stores the most recently predicted candidate token at each generation position, $\mathbf{c}^{(t)}$ stores raw model confidence, and $\widetilde{\mathbf{c}}^{(t)}$ stores history-aware confidence. A forward pass on $\mathbf{x}^{(t)}$ produces candidates and raw confidences labeled with superscript $t-1$ because they are used to construct $\mathbf{x}^{(t-1)}$.

In both forward calls, arguments before the semicolon are computational inputs, while the two arguments after it specify the retained query and key sets, respectively. Both routines form the retained attention using queries and keys from the globally designated layer $\ell^\star$; this layer index is omitted from $\mathbf{Q}$, $\mathbf{K}$, and $\widetilde{\mathbf{A}}$ throughout to keep the notation compact. Each entry $\widetilde{A}^{(t)}_{j,i}$ is the *head-averaged attention weight* for query position j and key position i, as defined in Equation equation 5, and only the required slice of these weights is stored. In every subscript of $\widetilde{\mathbf{A}}^{(t)}$, the first position set gives the global query indices and the second gives the global key indices. The block $\widetilde{\mathbf{A}}^{(t+1)}_{\mathcal{D}^{(t+1)},\mathcal{W}^{(t)}}$ used in Equation equation 6, whose query and key index sets are $\mathcal{D}^{(t+1)}$ and $\mathcal{W}^{(t)}$, respectively, is read from the previously retained slice and is not stored separately.

The extended window $\mathcal{W}^{(t)}_{\text{ext}}$ is introduced as an implementation detail. At step t, the exact positions whose attention weights will be needed to construct the mask set for the next window are not yet known, since they depend on the current commitment decision. We therefore retain attention weights over a conservative range that covers the worst-case advance, namely, the case in which the current window commits k tokens. Thus, $\mathcal{W}^{(t)}_{\text{ext}}$ is defined by extending the current window by k positions to the right, thereby covering the at most k new positions that may enter the next window $\mathcal{W}^{(t-1)}$.

FULLFORWARD recomputes the entire sequence. It returns candidate tokens and raw confidences on $\mathcal{W}^{(t)}$, the head-averaged attention slice with query indices in $\mathcal{W}^{(t)}$ and key indices in $\mathcal{W}^{(t)}_{\text{ext}}$, and the full KV cache $\mathcal{C}^{(t)}$.

SPARSEFORWARD recomputes only $\mathcal{A}^{(t)}$ while reusing $\mathcal{C}^{(t+1)}$ elsewhere. It returns candidate tokens and raw confidences on $\mathcal{M}^{(t)}$, the head-averaged attention slice with query indices in $\mathcal{M}^{(t)}$ and key indices in $\mathcal{W}^{(t)}_{\text{ext}}$, and the recomputed partial KV states $\mathcal{C}^{(t)}_{\mathcal{A}^{(t)}}$. Replacing only the corresponding entries of $\mathcal{C}^{(t+1)}$ with these partial states forms the full cache $\mathcal{C}^{(t)}$.

A.2 STORAGE OVERHEAD AND COMPUTATION ANALYSIS

We analyze the storage overhead and computational complexity of vanilla diffusion decoding, Fast-dLLM with DualCache, and HiAda during text generation. Let N denote the sequence length, B the fixed block size used by Fast-dLLM, k the active-set budget of HiAda, L_w the adaptive window size, and n_{F} and n_{H} the average numbers of tokens resolved per denoising step by Fast-dLLM and HiAda, respectively. In practice, $B, k, L_w \ll N$. We suppress constants associated with the number of Transformer layers and attention heads, while retaining the hidden dimension

in the FLOP expressions. We separately analyze attention-activation memory, cumulative query-token evaluations, token-wise dense FLOPs, and attention FLOPs; these quantities capture different aspects of inference cost and should not be interpreted interchangeably.

Memory complexity. A vanilla diffusion language model performs a full-sequence forward pass at every denoising step. Consequently, N query tokens attend to N key-value tokens, resulting in an attention matrix of size $N \times N$. The corresponding attention-activation memory complexity is therefore

$$\mathcal{O}(N^2).$$

Fast-dLLM adopts block-wise approximate KV caching. With DualCache, both the decoded prefix and the masked suffix are cached, such that only the current block needs to be recomputed during regular decoding steps. For a block size B, at most B query positions are processed while attending to cached key-value representations spanning the full sequence. The resulting attention matrix has size at most $B \times N$, giving a regular-step attention-memory complexity of

$$\mathcal{O}(BN).$$

After completing a block, Fast-dLLM updates the KV cache for the full sequence. The corresponding block-boundary update processes all N positions and therefore has attention-memory complexity

$$\mathcal{O}(N^2).$$

Thus, DualCache reduces the attention memory of the regular block-decoding steps, but does not eliminate the quadratic worst-case memory incurred by full-sequence cache updates.

At a regular non-refresh step, HiAda processes only the active mask set and the active cache set. Each set contains at most k positions, so the number of query tokens processed by the Transformer is at most $2k$. These active queries attend to the cached key-value representations of the full sequence. The resulting attention matrix therefore has size at most $2k \times N$, giving an attention-memory complexity of

$$\mathcal{O}(kN).$$

Since $k \ll N$, the dominant attention activation memory during regular sparse steps is substantially reduced compared with the $\mathcal{O}(N^2)$ full-sequence baseline. Compared with the $\mathcal{O}(BN)$ regular-step memory of Fast-dLLM, HiAda replaces the fixed block-level query set with a finer-grained token-level active set.

Both Fast-dLLM and HiAda maintain cached KV states spanning the sequence, whose storage scales linearly with N for a fixed model architecture. HiAda additionally maintains historical information for attention-guided position selection and history-aware confidence accumulation. Specifically, the retained attention slice has size at most

$$k \times (k + L_w),$$

while the token-level historical state requires $\mathcal{O}(N)$ storage. The additional history-specific storage of HiAda is therefore

$$\mathcal{O}(k^2 + kL_w + N).$$

Since $k, L_w \ll N$, this additional persistent storage is lower order relative to the quadratic attention activations of a full-sequence forward pass.

HiAda periodically performs a full-sequence refresh to reconstruct up-to-date representations and KV states. Such a refresh processes all N positions and therefore has attention-memory complexity

$$\mathcal{O}(N^2).$$

Consequently, similar to Fast-dLLM, HiAda does not reduce the worst-case attention-memory requirement. Its memory advantage instead arises during the regular sparse steps, where the attention-memory complexity is reduced to $\mathcal{O}(kN)$.

Cumulative query-token evaluations and FLOP analysis. We distinguish the number of token positions recomputed by the Transformer from the corresponding floating-point operations. Let q_t denote the number of query positions recomputed at denoising step t, let d be the hidden dimension, and let d_{ff} be the feed-forward dimension. We define the cumulative number of query-token evaluations as

$$Q = \sum_{t=1}^{T} q_t.$$

This quantity measures how many token positions pass through token-wise linear projections and feed-forward layers, but it is not equivalent to total FLOPs. A selective forward pass with q_t active queries and cached key-value states for all N sequence positions incurs

$$\mathcal{O}(q_t N d)$$

attention FLOPs and

$$\mathcal{O}\big(q_t(d^2 + d d_{\mathrm{ff}})\big)$$

token-wise dense FLOPs. For compactness, we write $C_{\mathrm{tok}} = d^2 + d d_{\mathrm{ff}}$ below.

Vanilla diffusion decoding. Under the standard schedule considered here, generating a sequence of length N uses N denoising steps, and every step recomputes all N query positions. The cumulative query-token evaluations are therefore

$$Q_{\mathrm{vanilla}} = N \cdot N = \mathcal{O}(N^2).$$

The associated token-wise dense cost is

$$\mathcal{O}(N^2 C_{\mathrm{tok}}).$$

Because every query attends to all N key-value positions, each step incurs $\mathcal{O}(N^2 d)$ attention FLOPs, giving the cumulative attention cost

$$C_{\mathrm{attn}}^{\mathrm{vanilla}} = N \cdot \mathcal{O}(N^2 d) = \mathcal{O}(N^3 d).$$

Fast-dLLM with DualCache. Fast-dLLM partitions the sequence into approximately N/B blocks. Let n_{F} denote the average number of tokens resolved per denoising step. Completing one block requires approximately B/n_{F} steps, and a regular step recomputes B query positions. Hence, regular block decoding contributes

$$Q_{\mathrm{reg}}^{\mathrm{F}} = \frac{N}{B} \cdot \frac{B}{n_{\mathrm{F}}} \cdot B = \mathcal{O}\left(\frac{BN}{n_{\mathrm{F}}}\right)$$

query-token evaluations. At each block boundary, Fast-dLLM performs a full-sequence cache update. Across approximately N/B blocks, these updates contribute

$$Q_{\mathrm{refresh}}^{\mathrm{F}} = \frac{N}{B} \cdot N = \mathcal{O}\left(\frac{N^2}{B}\right).$$

Therefore,

$$Q_{\mathrm{F}} = \mathcal{O}\left(\frac{BN}{n_{\mathrm{F}}} + \frac{N^2}{B}\right),$$

and the token-wise dense cost is

$$\mathcal{O}\left[\left(\frac{BN}{n_{\mathrm{F}}} + \frac{N^2}{B}\right) C_{\mathrm{tok}}\right].$$

Since both regular block queries and full-refresh queries attend to N key-value positions, the cumulative attention cost is

$$C_{\mathrm{attn}}^{\mathrm{F}} = \mathcal{O}\left(\frac{BN^2 d}{n_{\mathrm{F}}} + \frac{N^3 d}{B}\right).$$

HiAda. Let n_{H} denote the average number of tokens resolved per step. The complete HiAda trajectory requires approximately

$$T_{\mathrm{H}} \approx \frac{N}{n_{\mathrm{H}}}$$

denoising steps. At a regular sparse step, the active mask set and active cache set contain at most k positions each, so HiAda recomputes at most $2k$ query positions. The sparse steps therefore contribute

$$Q^{\mathrm{H}}_{\mathrm{sparse}} = \mathcal{O}\left(\frac{kN}{n_{\mathrm{H}}}\right)$$

query-token evaluations.

Let F_{H} denote the actual number of full-sequence refreshes. Since each refresh recomputes N query positions, the total number of query-token evaluations is

$$Q_{\mathrm{H}} = \mathcal{O}\left(\frac{kN}{n_{\mathrm{H}}} + F_{\mathrm{H}}N\right).$$

The corresponding token-wise dense cost is

$$C^{\mathrm{H}}_{\mathrm{dense}} = \mathcal{O}\left[\left(\frac{kN}{n_{\mathrm{H}}} + F_{\mathrm{H}}N\right) C_{\mathrm{tok}}\right].$$

During a sparse step, the active queries still attend to cached key-value states for all N positions. Consequently, sparse attention contributes $\mathcal{O}(kN^2d/n_{\mathrm{H}})$ FLOPs, while the full refreshes contribute $\mathcal{O}(F_{\mathrm{H}}N^2d)$. The cumulative attention cost is therefore

$$C^{\mathrm{H}}_{\mathrm{attn}} = \mathcal{O}\left(\frac{kN^2d}{n_{\mathrm{H}}} + F_{\mathrm{H}}N^2d\right).$$

For the fixed-refresh HiAda configuration, the refresh interval is

$$R = \max\left(1, \left\lfloor \frac{N}{k} \right\rfloor\right).$$

Since $T_{\mathrm{H}} \approx N/n_{\mathrm{H}}$, the number of full refreshes is $F_{\mathrm{H}} \approx k/n_{\mathrm{H}}$ up to the initial refresh, rounding, and boundary effects. Thus, for fixed k and n_{H}, HiAda has linear cumulative query-token evaluations and token-wise dense computation in N, but quadratic cumulative attention FLOPs:

$$Q_{\mathrm{H}} = \mathcal{O}(N), \qquad C^{\mathrm{H}}_{\mathrm{dense}} = \mathcal{O}(NC_{\mathrm{tok}}), \qquad C^{\mathrm{H}}_{\mathrm{attn}} = \mathcal{O}(N^2d).$$

Compared with vanilla diffusion decoding under $T = N$, HiAda therefore reduces the sequence-length dependence of cumulative attention computation from cubic to quadratic and that of token-wise dense computation from quadratic to linear. This FLOP reduction is distinct from the memory result: occasional full-sequence refreshes still retain an $\mathcal{O}(N^2)$ peak attention-memory requirement.

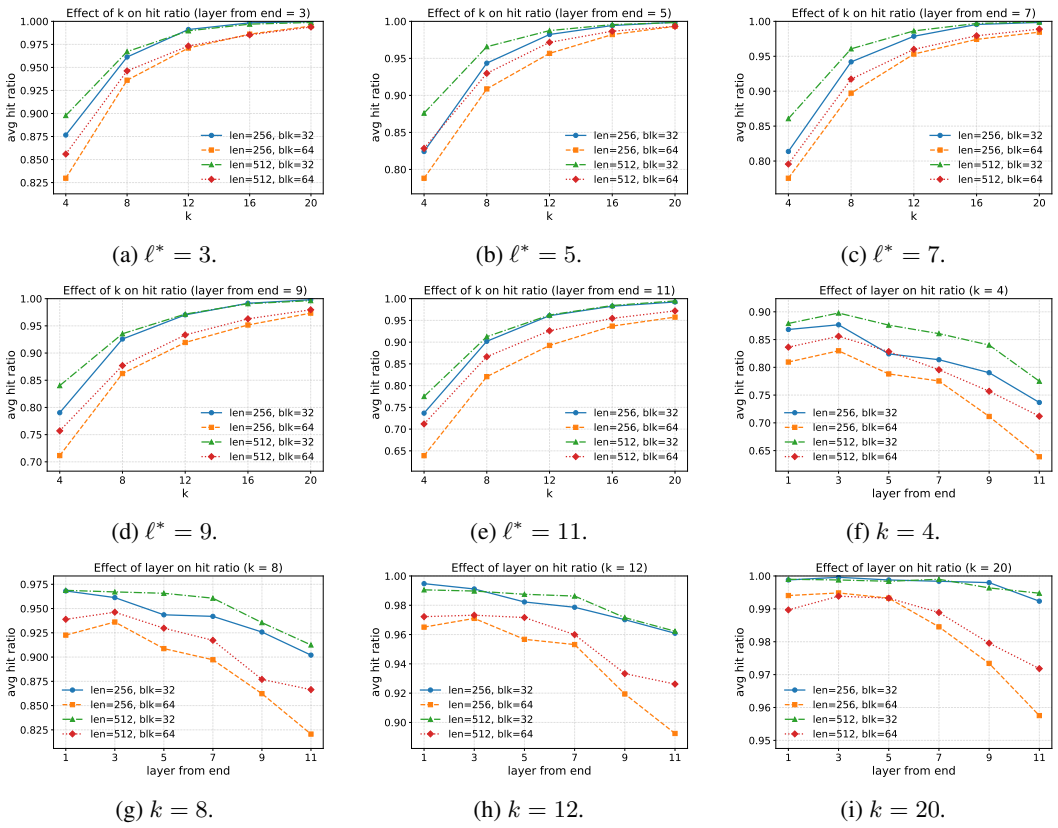
The preceding asymptotic simplification applies only to the fixed-refresh HiAda configuration. For the accuracy-oriented HiAda* schedule, the refresh interval changes with the number of remaining masked positions. Its cost is therefore parameterized by the resulting refresh count F^*_{H}:

$$Q_{\mathrm{H}^*} = \mathcal{O}\left(\frac{kN}{n_{\mathrm{H}}} + F^*_{\mathrm{H}}N\right), \qquad C^{\mathrm{H}^*}_{\mathrm{attn}} = \mathcal{O}\left(\frac{kN^2d}{n_{\mathrm{H}}} + F^*_{\mathrm{H}}N^2d\right).$$

We do not claim linear cumulative query-token scaling for HiAda* unless F^*_{H} is bounded independently of N.

A.3 Additional Hit-Ratio Experiments

To further evaluate the robustness of the attention-guided position predictor, we conduct additional hit-ratio experiments across a broader range of refresh frequencies and Transformer layers. Recall that the *hit ratio* measures the fraction of positions selected by the predicted active mask set that are actually decoded in the subsequent denoising step. These experiments complement Figure 3a–b by examining whether the predictive signal remains effective under different choices of the key attention layer ℓ^* and refresh divisor k.



(a) $\ell^* = 3$. (b) $\ell^* = 5$. (c) $\ell^* = 7$.

(d) $\ell^* = 9$. (e) $\ell^* = 11$. (f) $k = 4$.

(g) $k = 8$. (h) $k = 12$. (i) $k = 20$.

Figure 4: Additional analysis of the attention-guided position predictor. **(a)–(e)** Hit ratio as a function of the refresh divisor k when attention is extracted from different Transformer layers $\ell^* \in \{3, 5, 7, 9, 11\}$. **(f)–(i)** Hit ratio across Transformer layers under different refresh settings $k \in \{4, 8, 12, 20\}$. Together, these results evaluate the robustness of the historical attention signal with respect to both cache-refresh frequency and attention-layer selection.

Robustness across refresh frequencies. Figures 4(a)–(e) report the hit ratio obtained when the attention layer is fixed and the refresh divisor k is varied. Across different choices of ℓ^*, the attention-guided selector retains substantial predictive power over a broad range of refresh frequencies. Increasing k generally provides fresher attention information because full KV-cache refreshes are performed more frequently, which can improve the accuracy of predicting positions that will be decoded in the following step. At the same time, the improvement does not require refreshing the cache at every iteration, indicating that the attention signal remains informative even when it is reused across multiple sparse denoising steps. This further supports the periodic refresh strategy adopted by HiAda.

Robustness across Transformer layers. Figures 4(f)–(i) instead fix the refresh divisor and vary the Transformer layer from which attention is extracted. Although the absolute hit ratio varies across layers, useful predictive signals are observed across a broad range of layer choices and refresh frequencies. This suggests that the correlation between current attention patterns and future decoding positions is not restricted to a single particular layer. In practice, HiAda therefore requires attention statistics from only one designated layer rather than retaining attention maps from the entire Transformer, substantially reducing the additional storage required by the position predictor.

Overall observation. Taken together with Figure 3a–b, these additional results show that the attention-guided active-set construction is reasonably robust to both the refresh schedule and the choice of attention layer. More importantly, the results confirm that historical attention contains predictive information about subsequent decoding decisions even when the underlying KV states are not fully refreshed at every step. This property is central to HiAda: it enables the decoder to use

inexpensive historical signals to concentrate sparse computation on positions that are more likely to become useful in the next denoising iteration. "'

A.4 Detailed Experimental Parameters

We provide the inference hyperparameters used in our main experiments to facilitate reproducibility. All experiments are conducted on a server equipped with a single NVIDIA RTX A6000 GPU with 48 GB of GPU memory, using a batch size of 1. Unless otherwise specified, the same settings are used across GSM8K, MATH, HumanEval, and MBPP for a given model and decoding method.

LLaDA-8B-Instruct. For both HiAda and HiAda*, we set the adaptive mask-window size to $L_{\mathcal{W}} = 32$, the active-set budget to $k = 16$, the decoding confidence threshold to $\tau = 0.9$, and the history factor to $\gamma = 0.1$. The parameter k also controls the full KV-cache refresh frequency. HiAda adopts a fixed refresh interval

$$R = \max\left(1, \left\lfloor \frac{N}{k} \right\rfloor\right),$$

where N denotes the sequence length. HiAda* uses the same hyperparameters but dynamically adjusts the refresh interval according to the number of remaining masked tokens M, performing a full KV-cache refresh every M/k denoising steps. Consequently, full refreshes become progressively more frequent as decoding proceeds.

For Fast-dLLM, we use a fixed block size of 32 and a confidence threshold of $\tau = 0.9$. Unless otherwise specified, these settings are fixed across all benchmarks and generation lengths.

Dream-7B-Base. For both HiAda and HiAda*, we set the adaptive mask-window size to $L_{\mathcal{W}} = 32$, the active-set budget to $k = 16$, the decoding confidence threshold to $\tau = 0.9$, and the history factor to $\gamma = 0.1$. As with LLaDA-8B-Instruct, the parameter k also controls the full KV-cache refresh frequency. HiAda adopts the fixed refresh interval

$$R = \max\left(1, \left\lfloor \frac{N}{k} \right\rfloor\right),$$

whereas HiAda* dynamically adjusts the refresh interval according to the number of remaining masked tokens M, performing a full KV-cache refresh every M/k denoising steps.

For Fast-dLLM, we use a fixed block size of 32 and a confidence threshold of $\tau = 0.9$. Unless otherwise specified, these settings are fixed across all benchmarks and generation lengths.

A.5 KV_cache Similarity

Based on our observations, across different attention heads within the same layer, KV-cache updates exhibit highly similar patterns. In contrast, the update behavior changes systematically across layers and can be divided into three stages. In the first stage, substantial KV changes are concentrated at the newly decoded position and its nearby token positions. In the second stage, the changes extend from the decoding position to the subsequent token positions. In the final stage, KV values are updated across nearly all token positions. These observations reveal clear layer-dependent locality in KV-cache updates, while also indicating that a full cache refresh remains unavoidable in the later layers, where changes are distributed across the entire sequence.

A.6 Comparison with Other Acceleration Methods

We further compare HiAda with three representative training-free dLLM acceleration methods: Fast-dLLM, Dynamic-dLLM, and ES-dLLM. All experiments are conducted on a single NVIDIA A6000 GPU with a batch size of 1. We use a maximum generation length of 512 tokens for HumanEval and 256 tokens for GSM8K, MATH, and MBPP. The number of denoising steps is set equal to the maximum generation length. For methods using block-wise decoding, the block length is fixed to 32. All methods use deterministic decoding without classifier-free guidance. Unless otherwise specified, the remaining evaluation settings, including prompts and benchmark protocols, are identical to those used in the main experiments.

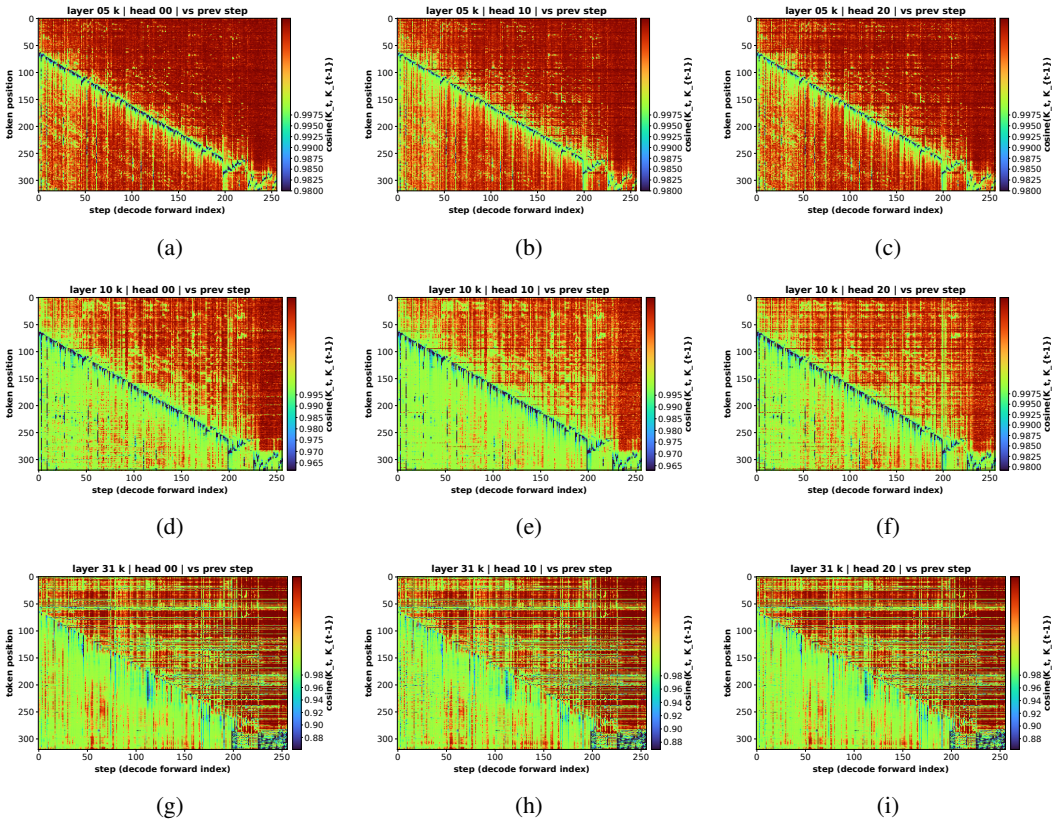


Figure 5: Visualization of K-cache dynamics in LLaDA-8B-Instruct. We compute the cosine similarity between the K-cache representations at corresponding token positions in two consecutive decoding steps, with a generation length of 256 and a block length of 32. Each row shows the similarity patterns of different attention heads within the same layer, while different rows correspond to different layers.

Fast-dLLM. We use the DualCache implementation of Fast-dLLM with a fixed block size of 32 and a confidence threshold of $\tau = 0.9$. The maximum generation length and the number of denoising steps are both set to 512 for HumanEval and 256 for GSM8K, MATH, and MBPP. All other settings follow the Fast-dLLM configuration used in our main experiments.

Dynamic-dLLM. We evaluate the complete Dynamic-dLLM algorithm, consisting of Dynamic Cache Updating (DCU) with feature caching and Adaptive Parallel Decoding (APD). For DCU, we enable feature caching (`is_feature_cache=True`), set the cache window size to 32 and the layer budget to 32, and select tokens according to their value states (`select_from=v`). For APD, we use `generate_mode=pd`, `pd_mode=2`, and a confidence threshold of 0.9. The APD coefficients are set to $\alpha = 0.001$ and $\beta = 0.0008$. We use a block length of 32, set the classifier-free guidance scale to zero, disable stochastic sampling, and use 512 denoising steps for HumanEval and 256 steps for the other benchmarks.

ES-dLLM. We evaluate the HIDDENSTATE variant of ES-dLLM, rather than its DualCache or no-cache variants. We set the early-skipping coefficient to $\alpha = 0.5$, with layer proportions $\{1, 0.5, 0.25\}$ applied at normalized decoding positions $\{0, 0.125, 0.25\}$. KV caching is enabled, while the optional parallel-decoding mode is disabled (`parallel_mode=False`). We use a block length of 32, set the classifier-free guidance scale to zero, and perform deterministic decoding with a temperature of zero. The maximum generation length is 512 for HumanEval and 256 for the other benchmarks.

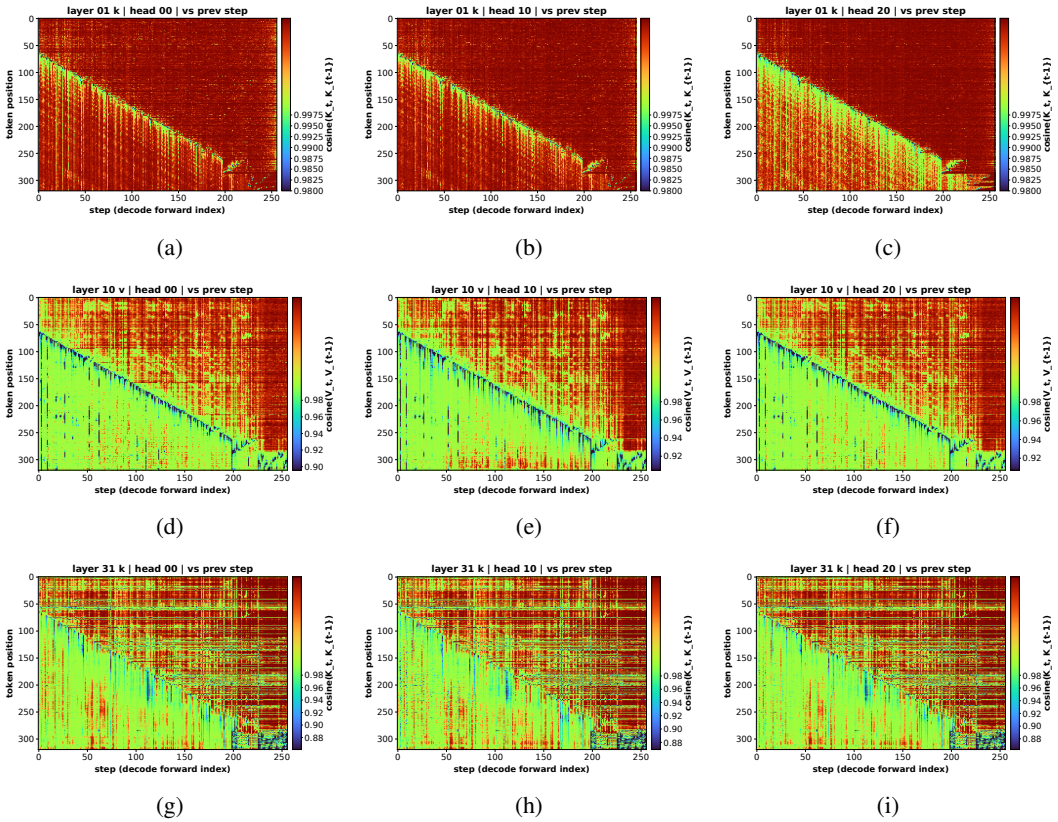


Figure 6: Visualization of V-cache dynamics in LLaDA-8B-Instruct. We compute the cosine similarity between the V-cache representations at corresponding token positions in two consecutive decoding steps, with a generation length of 256 and a block length of 32. Each row shows the similarity patterns of different attention heads within the same layer, while different rows correspond to different layers.

HiAda. We use the same HiAda configuration as in the main experiments. Specifically, the adaptive mask-window size is set to $L_{\mathcal{W}} = 32$, the active-set budget to $k = 16$, the decoding confidence threshold to $\tau = 0.9$, and the history factor to $\gamma = 0.1$. The fixed-refresh variant uses

$$R = \max\left(1, \left\lfloor \frac{N}{k} \right\rfloor\right),$$

where N denotes the sequence length. The maximum generation length and the number of denoising steps are both set to 512 for HumanEval and 256 for GSM8K, MATH, and MBPP.

Figure 7 compares the end-to-end efficiency of the four acceleration methods under aligned hardware, generation-length, and decoding settings.

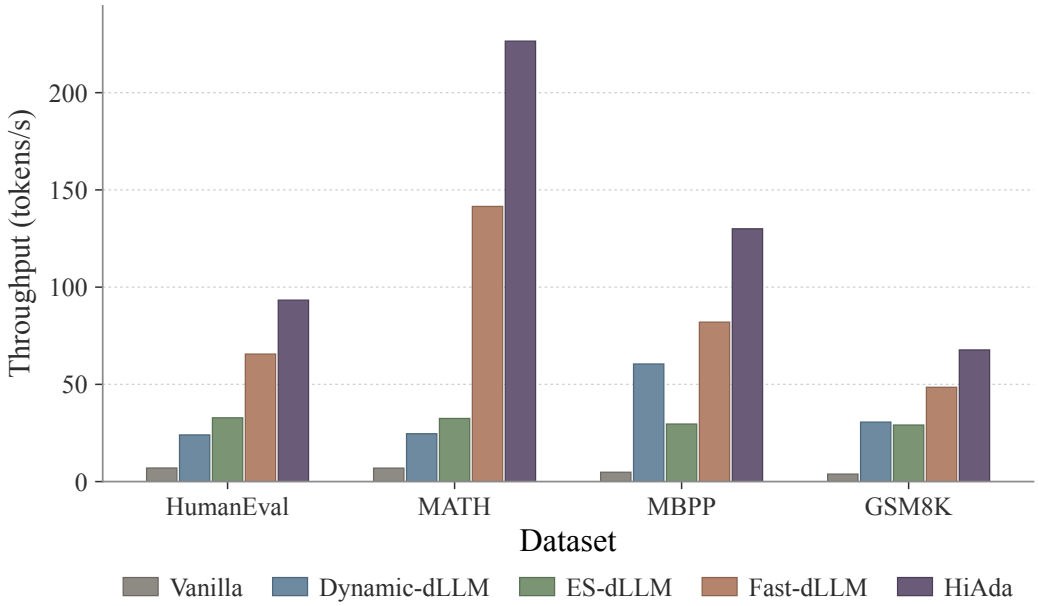


Figure 7: Comparison of HiAda with Fast-dLLM, Dynamic-dLLM, and ES-dLLM on GSM8K, MATH, HumanEval, and MBPP. All methods are evaluated on a single NVIDIA A6000 GPU with a batch size of 1 under matched benchmark-specific generation settings. The maximum generation length and number of denoising steps are set to 512 for HumanEval and 256 for the other benchmarks. Fast-dLLM uses DualCache, Dynamic-dLLM uses its complete DCU+APD configuration, and ES-dLLM uses the HIDDENSTATE variant with KV caching enabled and optional parallel decoding disabled. HiAda uses the same hyperparameters as in the main experiments.