Sparse-dLLM: Accelerating Diffusion LLMs with Dynamic Cache Eviction

Yuerong Song[1,2], **Xiaoran Liu**[1,2], **Ruixiao Li**[1,2], **Zhigeng Liu**[1,2], **Zengfeng Huang**[1,2], **Qipeng Guo**[2,3], **Ziwei He**[2*], **Xipeng Qiu**[1,2*]

[1]College of Computer Science and Artificial Intelligence, Fudan University
[2]Shanghai Innovation Institute
[3]Shanghai AI Lab
yuerongsong25@m.fudan.edu.cn, xpqiu@fudan.edu.cn, ziweihe@outlook.com

Abstract

Diffusion Large Language Models (dLLMs) enable breakthroughs in reasoning and parallel decoding but suffer from prohibitive quadratic computational complexity and memory overhead during inference. Current caching techniques accelerate decoding by storing full-layer states, yet impose substantial memory usage that limit long-context applications. Our analysis of attention patterns in dLLMs reveals persistent cross-layer sparsity, with pivotal tokens remaining salient across decoding steps and low-relevance tokens staying unimportant, motivating selective cache eviction. We propose Sparse-dLLM, the first training-free framework integrating dynamic cache eviction with sparse attention via delayed bidirectional sparse caching. By leveraging the stability of token saliency over steps, it retains critical tokens and dynamically evicts unimportant prefix/suffix entries using an attention-guided strategy. Extensive experiments on LLaDA and Dream series demonstrate Sparse-dLLM achieves up to 10 times higher throughput than vanilla dLLMs, with comparable performance and similar peak memory costs, outperforming previous methods in efficiency and effectiveness.

Code — https://github.com/OpenMOSS/Sparse-dLLM
Extended version — https://arxiv.org/abs/2508.02558

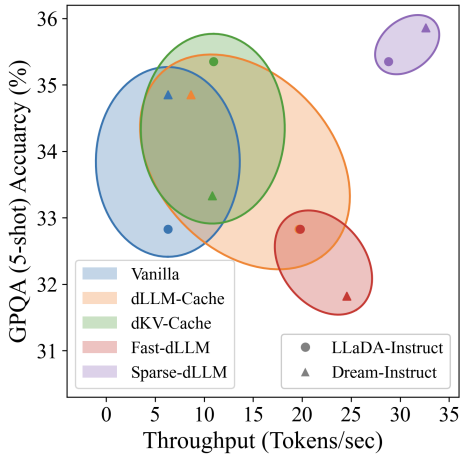


Figure 1: Throughput vs. Accuracy across methods. Sparse-dLLM (ours) achieves the best throughput while maintaining or even improving performance of vanilla dLLMs.

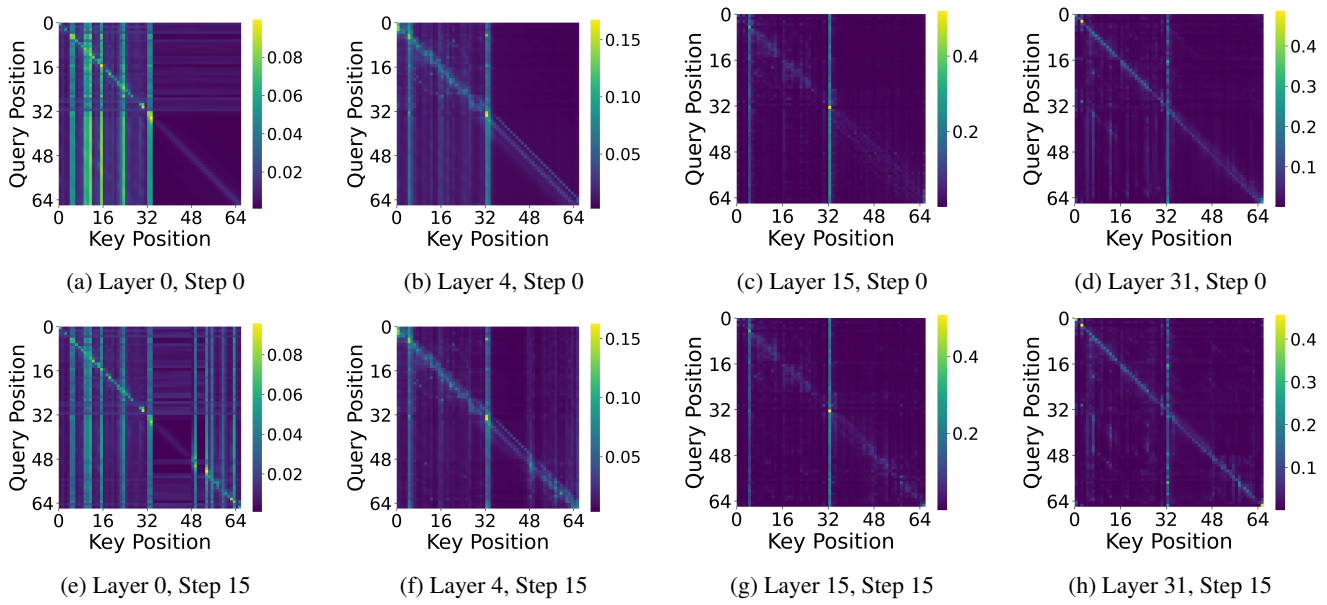
Introduction

Diffusion Large Language Models, or dLLMs, have garnered significant attention in the Natural Language Processing community (Nie et al. 2025; Ye et al. 2025). They are seen as a promising approach to addressing key limitations of traditional auto-regressive LLMs (Touvron et al. 2023; Sun et al. 2024), such as the reversal curse (Berglund et al. 2023), and enabling advanced reasoning (Dziri et al. 2023; Ye et al. 2025), and parallel decoding (Inception 2025; Gemini 2025). Extensive research has been dedicated to exploring their scalability (Nie et al. 2025; Ye et al. 2025), adapting them for multi-modal applications (Yang et al. 2025; You et al. 2025; Yu, Ma, and Wang 2025), and adapting them for reasoning tasks (Zhao et al. 2025; Huang et al. 2025; Zhu et al. 2025). However, current open-source dLLMs show a significant throughput shortage in practice, with their actual speed lags behind that of auto-regressive LLMs (Ma et al. 2025; Hu et al. 2025; Wu et al. 2025; Liu et al. 2025).

While traditional auto-regressive LLMs exhibit $\mathcal{O}(L)$ computational complexity during decoding (where L denotes the prompt length), dLLMs incur a significantly higher $\mathcal{O}(L^2)$ complexity. This stems from the requirement to recompute the QKV states for the entire sequence, including the input prompt, all generated tokens, and mask tokens at every inference step. To address this cost, recent studies have adapted the KV cache mechanism from auto-regressive LLMs to dLLMs (Ma et al. 2025; Wu et al. 2025; Liu et al. 2025). These approaches leverage the observation that the KV states across consecutive decoding steps are often nearly identical (Ma et al. 2025; Liu et al. 2025). Consequently, they allow multiple steps to reuse the same cached KV states, accelerating decoding without compromising output quality. While this reuse strategy achieves computational savings by storing the complete KV representations for the sequence across all layers, this substantial memory overhead hinders the practical deployment of dLLMs for long-context scenarios.

Motivated by this limitation, we conduct a thorough anal-

* Corresponding Author.



(a) Layer 0, Step 0 (b) Layer 4, Step 0 (c) Layer 15, Step 0 (d) Layer 31, Step 0

(e) Layer 0, Step 15 (f) Layer 4, Step 15 (g) Layer 15, Step 15 (h) Layer 31, Step 15

Figure 2: Sparsity patterns in dLLM attention. Using LLaDA-8B-Instruct with L = 66, T = 32, and a block length of 32, we observe pronounced sparsity that persists across layers, with pivotal tokens remaining salient throughout decoding steps.

ysis of attention patterns in dLLMs. As shown in Figure 2, dLLMs exhibit significant sparsity, akin to auto-regressive LLMs, characterized by a sharp concentration on local positions and the vertical attention pattern (Jiang et al. 2024) (where certain pivotal tokens remain salient across queries and decoding steps). Note that, unlike the causal attention of auto-regressive LLMs, dLLMs' bidirectional attention does not exhibit abnormal focus on initial tokens (Xiao et al. 2023). Crucially, we find that although QKV states are recomputed at every inference step, the specific ones receiving significant attention remain remarkably stable across steps, indicating that low-salience tokens identified in early steps persistently exhibit minimal relevance throughout decoding. These observations—sparsity and stable key tokens—motivate our strategy: selectively evicting unimportant KV cache entries while retaining only the critical subset in early steps at each layer. Our approach markedly improves dLLM computational efficiency while introducing only minimal additional memory overhead, and without degrading downstream performance.

We introduce **Sparse-dLLM**: the first training-free framework to integrate dynamic cache eviction with sparse attention for dLLMs. As illustrated in Figure 3, our approach employs a delayed bidirectional sparse strategy. First, we cache KV states for tokens outside the current decoding block during early inference step. Notably, we delay cache updates by one step to ensure stability. Then, leveraging temporal consistency in token saliency, we dynamically evict low-importance KV entries for both prefix and suffix tokens, guided by attention-aware sparse patterns with a pre-defined retention ratio. Cache states are fully refreshed when transitioning between blocks (Wu et al. 2025). As evidenced in Figure 1, Sparse-dLLM achieves the best throughput while maintaining or even enhancing performance of vanilla dLLMs on certain tasks. Critically, by pruning redundant cache entries, our method maintains near-identical peak memory costs to vanilla dLLMs while optimizing computational efficiency. Our contributions are:

- We establish a formal analysis of sparsity patterns in dLLMs, revealing persistent cross-layer attention sparsity, with pivotal tokens remaining salient across decoding steps and low-relevance tokens staying unimportant, motivating our selective cache eviction strategy.
- We propose Sparse-dLLM, the first training-free dynamic cache eviction method for dLLMs, featuring novel delayed bidirectional sparse caching that enables plug-and-play inference acceleration.
- Extensive experiments on LLaDA and Dream demonstrate Sparse-dLLM achieves up to 10× higher throughput than vanilla dLLMs, while maintaining comparable performance and nearly identical memory costs, exceeding previous methods in efficiency and effectiveness.

Related Work

KV cache optimization is critical for auto-regressive LLMs. Causal attention in AR maintains KV states for the input and generated tokens, allowing for direct caching and trading memory for computation. However, cache size grows with input length, limiting long-context deployment and driving KV cache optimization, whose typical strategy is token eviction. Current methods for KV cache sparsification in auto-regressive LLMs are retrospective, using fixed rules (Xiao et al. 2023), past attention scores (Zhang et al. 2024; Ge et al. 2023), or filtering based on a portion of previous tokens like SnapKV (Li et al. 2025), to manage already-generated tokens. Unlike auto-regressive LLMs, which can only see previously generated tokens, dLLMs can see the complete

sequence. Therefore, our method designed for dLLMs retrospectively and prospectively sparsifies the cache, considering both prefix and suffix entries. These entries represent the cache states for the tokens preceding and succeeding the current block, respectively.

Although bidirectional attention in dLLMs prevents direct caching, recent studies leverage the observation that KV states across consecutive decoding steps are often nearly identical. Consequently, they adapt the KV cache mechanism from auto-regressive LLMs to dLLMs, which accelerates decoding without compromising output quality, as in dLLM-Cache (Liu et al. 2025), dKV-Cache (Ma et al. 2025), FreeCache (Hu et al. 2025), and Fast-dLLM (Wu et al. 2025). Specifically, dLLM-Cache (Liu et al. 2025) sets different refresh intervals for prompt cache and response cache, and uses feature similarity to update response partially. dKV-Cache (Ma et al. 2025) implements one-step delayed caching, where decoded tokens are cached not at their current decoding step but at the subsequent step, combined with a refreshing mechanism. Based on the rapidly diminishing contribution from masked tokens to earlier unmasked tokens, FreeCache (Hu et al. 2025) caches prompt tokens' KV states. Fast-dLLM (Wu et al. 2025) caches all KV states excluding the current decoding block. However, these approaches merely introduce a KV cache in dLLMs without examining its internal properties or further sparsifying it.

Method

Preliminary: Inference of dLLM

Unlike auto-regressive LLMs, diffusion language models (dLLMs) employ an iterative unmasking process to generate text through T discrete decoding steps, progressively transforming a fully masked initial sequence into the final output. Using LLaDA (Nie et al. 2025) as an example, we formalize this process as follows.

Let $\mathcal{V}$ be the vocabulary, $[\text{MASK}] \in \mathcal{V}$ be the special mask token, $\mathbf{x}^t \in \mathcal{V}^L$ be the sequence state at step t for $t = T, \cdots, 0$. The initial state is defined as:

$$\mathbf{x}^T = (\mathbf{c}_0, \cdots, \mathbf{c}_{p-1}, [\text{MASK}], \cdots, [\text{MASK}])$$

where $(\mathbf{c}_0, \cdots, \mathbf{c}_{p-1})$ constitute the prompt and L denotes the total sequence length with $L - p$ mask tokens.

At each step $t = T, \cdots, 1$, a mask predictor model f_θ computes logits for the entire sequence:

$$\mathbf{z}^t = f_\theta(\mathbf{x}^t)$$

Then, greedy decoding is performed on $\mathbf{z}^t$ to derive the predicted tokens $\hat{\mathbf{x}}^t$ for all masked positions:

$$\hat{\mathbf{x}}_i^t = \arg\max_{v \in \mathcal{V}} (\text{Softmax}(\mathbf{z}_i^t))_v \quad \text{if } \mathbf{x}_i^t == [\text{MASK}]$$

Finally, The transition function S (Liu et al. 2025) selectively updates tokens in $\mathbf{x}^t$ based on predicted tokens $\hat{\mathbf{x}}^t$ (e.g., random or by confidence) to generate $\mathbf{x}^{t-1}$:

$$\mathbf{x}^{t-1} = S(\hat{\mathbf{x}}^t, \mathbf{x}^t)$$

After T steps, the final generated sequence $\mathbf{x}^0$ contains no mask tokens. However, iterative recomputation of all attention states for the full sequence imposes a substantial computational overhead, markedly increasing inference latency.

Observations

To unlock the potential of KV cache optimization for memory-efficient dLLM inference acceleration, we begin by systematically analyzing the attention patterns. As shown in Figure 2, our analysis reveals two fundamental features:

Sparsity Across Layers Horizontally in Figure 2, we observe consistent sparsity across all dLLM layers within individual steps. Unlike auto-regressive models, dLLMs show no abnormal initial-token focus, but exhibit two stable patterns: (1) Local attention (bright diagonals) with strong neighbor focus, and (2) Vertical attention (bright verticals) where all queries concentrate on few pivotal keys. These patterns persist uniformly across layers, with most positions receiving minimal weights.

Consistency Across Steps Vertically in Figure 2, attention patterns across inference steps reveal remarkable temporal consistency in token saliency for tokens outside the current block. Take attention maps in Layer 0 as an example, as shown in Figures 2a and 2e. Although QKV states are recomputed at every inference step, the specific tokens receiving significant attention remain remarkably stable across steps. This suggests that low-salience tokens outside the current block, once identified, consistently show minimal relevance throughout the decoding process.

These observations motivate us to selectively retain only the critical entries in the KV cache while evicting unimportant ones. Notably, while the local attention pattern is stable, we do not specifically use it for KV cache optimization, as its effective window in dLLMs is much smaller than the block length, rendering such optimizations unnecessary.

Sparse-dLLM

We propose Sparse-dLLM, the first training-free framework to integrate a dynamic bidirectional cache eviction with sparse attention for dLLMs. Specifically, our method introduces two main strategies to manage the KV cache: (1) Dynamic bidirectional cache eviction, which leverages the temporal consistency in token saliency by using attention-aware sparse patterns to dynamically evict low-importance KV entries from both the prefix and suffix tokens (i.e., tokens preceding and succeeding the current block). (2) Delayed cache updates, where cache updates are intentionally delayed by one step to improve stability. The cache is fully cleared and refreshed when moving to a new decoding block.

Dynamic Bidirectional Cache Eviction In contrast to auto-regressive LLMs that only sparsify prefix tokens, Sparse-dLLM widens the scope of cache eviction, targeting tokens from both the prefix (those preceding the current block) and the suffix (those succeeding the current block). Let b denote the block length, p denote the length of the prompt, and $o \in [p, L)$ denote the positional offset of the first token in the current block. Then the candidate set $\mathbf{K}_f$, $\mathbf{V}_f$ from the KV states outside the current block are:

$$\mathbf{K}_f = \text{Concat}[\mathbf{K}_{:o}, \mathbf{K}_{o+b:}], \quad \mathbf{V}_f = \text{Concat}[\mathbf{V}_{:o}, \mathbf{V}_{o+b:}]$$

Through observations from Figure 2, we identify consistency in token saliency across queries and steps. This insight

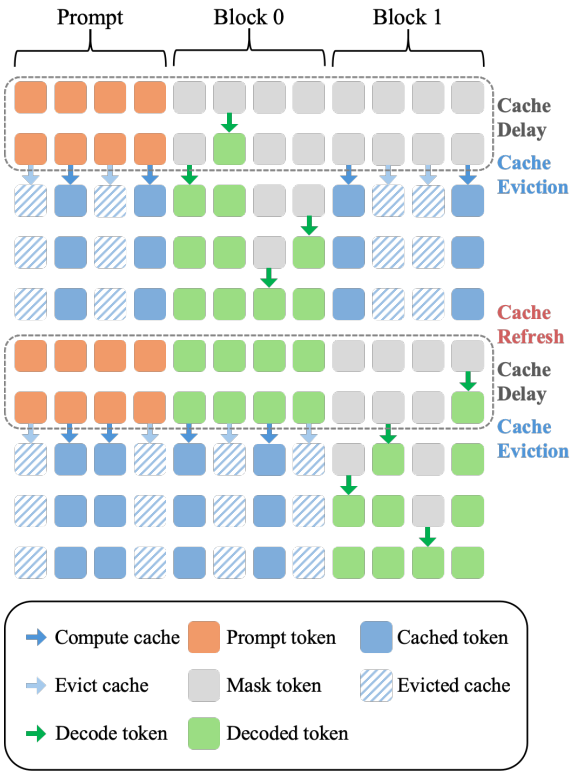


Figure 3: Overview of Sparse-dLLM.

enables us to directly sparsify the cache by computing attention scores between the current block's query states and the candidate K states. For the current block's query states $\mathbf{Q}_b$, the attention scores are computed as:

$$\mathbf{A} = \frac{\mathbf{Q}_b \mathbf{K}_f^T}{\sqrt{d_k}}.$$

Drawing inspiration from SnapKV (Li et al. 2025), we incorporate max pooling operation to aggregate local information. This design prevents potential performance degradation caused by incomplete data when only partial details are preserved after dynamic cache eviction. Let $r \in [0, 1]$ represent the retention ratio, s represent the kernel size for max pooling. We can derive the indices of pivotal tokens through top-k selection, where $k = (L - b) \times r$:

$$\text{Indices} = \text{top-}k(\text{MaxPool}(\mathbf{A})).$$

The final KV cache $\mathbf{K}_c$, $\mathbf{V}_c$ is then constructed as:

$$\mathbf{K}_c = \mathbf{K}_f[\text{Indices}], \quad \mathbf{V}_c = \mathbf{V}_f[\text{Indices}]$$

Through its dynamic bidirectional cache eviction strategy, Sparse-dLLM effectively reduces the number of KV cache entries, which in turn decreases memory consumption while boosting inference throughput.

Delayed Cache Updates Furthermore, by observing the L2-norm of KV state changes outside the current decoding block between adjacent steps, as illustrated in Figure 4, it

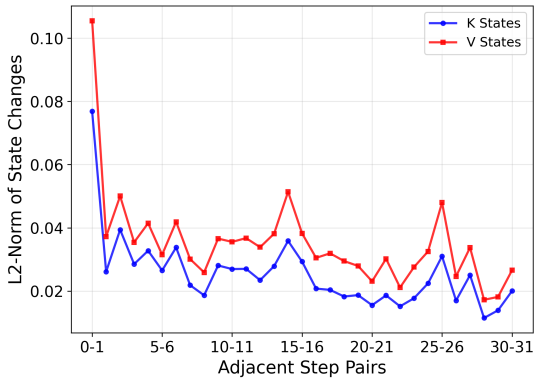


Figure 4: L2-norm of KV state changes outside the current decoding block between adjacent step pairs.

can be noted that the variation in KV states is relatively significant between step 0 and step 1. This observation suggests that the KV states intended for caching may not yet have stabilized at step 0 of the decoding block. Therefore, we delay the KV cache updates by one step upon decoding each block to mitigate early-stage instability in cached KV states.

Experiment

Setup

We conduct experiments on the existing dLLMs, including LLaDA-8B-Instruct (Nie et al. 2025), LLaDA-1.5 (Zhu et al. 2025), Dream-v0-7B-Base, and Dream-v0-7B-Instruct (Ye et al. 2025). By default, we set the block length to 32 and keep the unmasking strategy in the official code of LLaDA and Dream. For all models, we apply a fixed random seed 2025, retention ratio $r = 0.5$, and kernel size $s = 3$.

The evaluation metrics include accuracy, throughput (Tokens Per Second, TPS), and peak memory (GB). We use OpenCompass (Contributors 2023) to benchmark performance on general, science, math, and code tasks: MMLU (5-shot) (Hendrycks et al. 2020), ARC-c (0-shot) (Clark et al. 2018), PIQA (0-shot) (Bisk et al. 2020), GPQA (5-shot) (Rein et al. 2024), GSM8K (4-shot) (Cobbe et al. 2021), Math (4-shot) (Hendrycks et al. 2021), and HumanEval (0-shot) (Chen et al. 2021). All experiments were run on NVIDIA 4090 (48GB) GPUs. Note that our implementation is based on the native dLLM codebase, as popular inference frameworks like vLLM (Kwon et al. 2023) are optimized for auto-regressive models and would require substantial adaptation for dLLMs.

For performance, we report the mean score over three independent trials. For efficiency, we average results from ten randomly sampled instances, consistent across all methods.

Main Results

The main results comparing the baseline, other methods and our proposed Sparse-dLLM on LLaDA and Dream are presented in Table 1 and Table 2, respectively. These results demonstrate that our Sparse-dLLM method achieves

	MMLU		ARC-C		PIQA		GPQA		GSM8K		Math		HE		Avg.
LLaDA-8B-Instruct	60.60		**88.47**		83.62		32.83		78.39		36.02		34.76		59.24
Throughput (TPS, ↑)	9.48	1.0×	21.08	1.0×	21.74	1.0×	6.30	1.0×	4.57	1.0×	8.52	1.0×	12.53	1.0×	1.0×
Memory (GB, ↓)	**15.54**	**1.0**×	**15.18**	**1.0**×	**15.17**	**1.0**×	**15.86**	**1.0**×	**16.18**	**1.0**×	**15.63**	**1.0**×	**15.39**	**1.0**×	**1.0**×
+ dLLM-Cache	61.40		87.46		83.62		32.83		78.92		**36.56**		**37.80**		59.80
Throughput (TPS, ↑)	21.43	2.3×	24.03	1.1×	23.64	1.1×	19.68	3.1×	18.03	3.9×	21.11	2.5×	21.64	1.7×	2.3×
Memory (GB, ↓)	16.61	1.1×	15.82	1.0×	15.80	1.0×	17.19	1.1×	17.85	1.1×	16.74	1.1×	16.65	1.1×	1.1×
+ dKV-Cache	60.87		87.80		83.73		**35.35**		**79.30**		35.46		37.20		**59.96**
Throughput (TPS, ↑)	14.34	1.5×	18.28	0.9×	18.34	0.8×	10.95	1.7×	8.89	1.9×	13.46	1.6×	14.40	1.1×	1.4×
Memory (GB, ↓)	17.88	1.2×	16.10	1.1×	16.05	1.1×	19.46	1.2×	21.08	1.3×	18.34	1.2×	17.17	1.1×	1.2×
+ Fast-dLLM	**61.43**		87.80		83.79		32.83		75.89		33.78		36.59		58.87
Throughput (TPS, ↑)	20.51	2.2×	21.63	1.0×	21.99	1.0×	19.82	3.1×	19.11	4.2×	20.72	2.4×	21.50	1.7×	2.2×
Memory (GB, ↓)	17.13	1.1×	15.81	1.0×	15.77	1.0×	18.29	1.2×	19.48	1.2×	17.47	1.1×	16.60	1.1×	1.1×
+ Sparse-dLLM (ours)	61.01		**88.47**		**84.44**		**35.35**		77.56		34.42		**37.80**		59.86
Throughput (TPS, ↑)	**31.89**	**3.4**×	**36.85**	**1.7**×	**37.05**	**1.7**×	**28.82**	**4.6**×	**26.45**	**5.8**×	**31.72**	**3.7**×	**34.39**	**2.7**×	**3.4**×
Memory (GB, ↓)	15.73	1.0×	15.26	1.0×	15.24	1.0×	16.16	1.0×	16.60	1.0×	15.86	1.0×	15.54	1.0×	1.0×
LLaDA-1.5	61.05		**88.47**		83.79		33.33		81.35		38.04		**40.24**		60.90
Throughput (TPS, ↑)	9.46	1.0×	21.10	1.0×	21.75	1.0×	6.30	1.0×	4.57	1.0×	8.52	1.0×	12.52	1.0×	1.0×
Memory (GB, ↓)	**15.54**	**1.0**×	**15.18**	**1.0**×	**15.17**	**1.0**×	**15.86**	**1.0**×	**16.18**	**1.0**×	**15.63**	**1.0**×	**15.39**	**1.0**×	**1.0**×
+ dLLM-Cache	61.41		**88.47**		83.62		32.83		81.65		37.04		37.80		60.40
Throughput (TPS, ↑)	21.40	2.3×	23.95	1.1×	24.85	1.1×	20.41	3.2×	18.57	4.1×	21.72	2.6×	21.23	1.7×	2.3×
Memory (GB, ↓)	16.61	1.1×	15.83	1.0×	15.80	1.0×	17.19	1.1×	17.85	1.1×	16.74	1.1×	16.65	1.1×	1.1×
+ dKV-Cache	61.34		88.14		84.28		33.33		**82.34**		**38.08**		39.63		**61.02**
Throughput (TPS, ↑)	14.29	1.5×	18.26	0.9×	18.41	0.8×	10.98	1.7×	8.91	2.0×	13.62	1.6×	14.40	1.2×	1.4×
Memory (GB, ↓)	17.88	1.2×	16.10	1.1×	16.05	1.1×	19.46	1.2×	21.08	1.3×	18.34	1.2×	17.17	1.1×	1.2×
+ Fast-dLLM	**61.57**		88.14		83.95		31.82		80.82		36.60		36.59		59.93
Throughput (TPS, ↑)	20.74	2.2×	21.89	1.0×	21.67	1.0×	20.09	3.2×	19.60	4.3×	20.59	2.4×	21.36	1.7×	2.3×
Memory (GB, ↓)	17.13	1.1×	15.81	1.0×	15.77	1.0×	18.29	1.2×	19.48	1.2×	17.47	1.1×	16.60	1.1×	1.1×
+ Sparse-dLLM (ours)	61.37		88.14		**84.71**		**34.34**		81.43		37.32		39.63		60.99
Throughput (TPS, ↑)	**32.05**	**3.4**×	**36.78**	**1.7**×	**36.96**	**1.7**×	**29.06**	**4.6**×	**26.21**	**5.7**×	**31.87**	**3.7**×	**34.18**	**2.7**×	**3.4**×
Memory (GB, ↓)	15.73	1.0×	15.26	1.0×	15.24	1.0×	16.16	1.0×	16.60	1.0×	15.86	1.0×	15.54	1.0×	1.0×

Table 1: Comprehensive benchmark results on LLaDA-8B-Instruct (Nie et al. 2025) and LLaDA-1.5 (Zhu et al. 2025). Each cell presents the accuracy, decoding throughput in tokens per second and peak memory cost in GB with relative efficiency to the pre-trained model. Best values in bold, suboptimal values underlined.

the most significant throughput improvement while maintaining or even slightly enhancing performance, with nearly identical peak memory to vanilla dLLMs.

Sparse-dLLM delivers a remarkable leap in throughput. For instance, on the GSM8K dataset with LLaDA-8B-Instruct, it boosts throughput from 4.57 to 26.45 TPS (a 5.8× speedup). Similarly, with Dream-v0-7B-Instruct on the GPQA dataset, throughput increases from 6.29 to 32.63 TPS (a 5.2× acceleration). Across all evaluated configurations, Sparse-dLLM consistently achieves the highest throughput.

Sparse-dLLM maintains memory consumption comparable to vanilla dLLMs, a stark contrast to other methods that incur significant memory overhead. Specifically, on LLaDA models, its peak memory differs by less than 0.5 GB from the baseline. More significantly, the block-wise decoding introduced on Dream allows the sampling process to use only the logits from the current block. As a result, Sparse-dLLM's memory usage remains even lower than the baseline, despite the introduction of KV cache.

Sparse-dLLM attains its efficiency improvements while maintaining high performance fidelity. On LLaDA models, it not only matches but also delivers a marginal gain in accuracy. A notable exception, however, occurs on the HumanEval benchmark with Dream models. Our analysis suggests this performance drop is attributable to our general-purpose eviction strategy removing syntactically crucial tokens (e.g., variable types, numeric literals), which impairs the model's code generation ability.

Long-Context Results

Our evaluation demonstrates Sparse-dLLM's comprehensive advantages in long-context scenarios.

As shown in Figure 5, Sparse-dLLM consistently outperforms other methods, achieving up to 10× higher throughput than the vanilla dLLM at a 4k sequence length. Its peak memory consumption remains remarkably low with a nearly flat growth curve, in sharp contrast to competitors like Fast-dLLM which suffer from Out-of-Memory (OOM) errors on

	MMLU		ARC-c		PIQA		GPQA		GSM8K		Math		HE		Avg.
Dream-v0-7B-Base	**72.96**		82.71		81.18		**32.83**		70.74		20.78		**53.05**		59.18
Throughput (TPS, ↑)	9.88	1.0×	19.60	1.0×	20.15	1.0×	6.33	1.0×	7.38	1.0×	8.93	1.0×	11.80	1.0×	1.0×
Memory (GB, ↓)	15.64	1.0×	15.49	1.0×	15.49	1.0×	15.77	1.0×	15.73	1.0×	15.67	1.0×	16.73	1.0×	1.0×
+ dLLM-Cache	72.87		85.08		81.12		31.31		72.55		20.40		50.61		59.13
Throughput (TPS, ↑)	12.68	1.3×	19.33	1.0×	19.75	1.0×	8.72	1.4×	9.87	1.3×	12.16	1.4×	14.24	1.2×	1.2×
Memory (GB, ↓)	16.37	1.0×	15.79	1.0×	15.77	1.0×	16.93	1.1×	16.76	1.1×	16.50	1.1×	17.28	1.0×	1.0×
+ dKV-Cache	72.77		82.03		81.39		**32.83**		69.90		20.06		45.73		57.82
Throughput (TPS, ↑)	13.90	1.4×	19.65	1.0×	19.86	1.0×	11.03	1.7×	11.91	1.6×	13.43	1.5×	13.14	1.1×	1.3×
Memory (GB, ↓)	15.92	1.0×	15.60	1.0×	15.59	1.0×	16.23	1.0×	16.14	1.0×	16.00	1.0×	16.95	1.0×	1.0×
+ Fast-dLLM	72.69		**86.78**		**82.86**		31.31		73.09		19.90		41.46		58.30
Throughput (TPS, ↑)	25.72	2.6×	27.18	1.4×	26.78	1.3×	24.26	3.8×	24.78	3.4×	25.39	2.8×	26.19	2.2×	2.5×
Memory (GB, ↓)	18.32	1.2×	15.85	1.0×	15.77	1.0×	20.69	1.3×	19.95	1.3×	18.94	1.2×	17.31	1.0×	1.1×
+ Sparse-dLLM (ours)	72.61		**86.78**		81.39		30.81		**74.15**		**23.60**		45.12		**59.21**
Throughput (TPS, ↑)	**36.97**	**3.7**×	**42.00**	**2.1**×	**42.27**	**2.1**×	**32.71**	**5.2**×	**34.28**	**4.6**×	**36.56**	**4.1**×	**38.91**	**3.3**×	**3.6**×
Memory (GB, ↓)	**14.74**	**0.9**×	**14.52**	**0.9**×	**14.52**	**0.9**×	**15.03**	**1.0**×	**14.94**	**0.9**×	**14.81**	**0.9**×	**14.62**	**0.9**×	**0.9**×
Dream-v0-7B-Instruct	72.42		**90.17**		88.25		34.85		76.57		39.38		57.93		**65.65**
Throughput (TPS, ↑)	9.61	1.0×	19.23	1.0×	19.65	1.0×	6.29	1.0×	7.27	1.0×	8.93	1.0×	11.41	1.0×	1.0×
Memory (GB, ↓)	15.64	1.0×	15.50	1.0×	15.49	1.0×	15.78	1.0×	15.73	1.0×	15.68	1.0×	16.74	1.0×	1.0×
+ dLLM-Cache	**72.55**		**90.17**		**88.79**		34.85		75.74		37.44		**59.15**		65.53
Throughput (TPS, ↑)	12.48	1.3×	19.54	1.0×	19.66	1.0×	8.66	1.4×	9.79	1.3×	12.11	1.4×	13.97	1.2×	1.2×
Memory (GB, ↓)	16.41	1.0×	15.80	1.0×	15.79	1.0×	16.96	1.1×	16.78	1.1×	16.52	1.1×	17.30	1.0×	1.0×
+ dKV-Cache	72.37		**90.17**		88.36		33.33		77.48		37.94		56.10		65.11
Throughput (TPS, ↑)	14.14	1.5×	19.44	1.0×	19.65	1.0×	10.82	1.7×	11.97	1.6×	13.33	1.5×	12.97	1.1×	1.4×
Memory (GB, ↓)	15.93	1.0×	15.61	1.0×	15.60	1.0×	16.24	1.0×	16.15	1.0×	16.01	1.0×	16.96	1.0×	1.0×
+ Fast-dLLM	70.81		89.83		86.02		31.82		77.79		39.82		52.44		64.08
Throughput (TPS, ↑)	25.37	2.6×	27.66	1.4×	27.75	1.4×	24.54	3.9×	24.57	3.4×	25.43	2.8×	26.15	2.3×	2.6×
Memory (GB, ↓)	18.41	1.2×	15.94	1.0×	15.86	1.0×	20.78	1.3×	20.03	1.3×	19.02	1.2×	17.39	1.0×	1.2×
+ Sparse-dLLM (ours)	70.94		88.47		86.62		**35.86**		**78.17**		**40.48**		50.00		64.36
Throughput (TPS, ↑)	**36.89**	**3.8**×	**42.01**	**2.2**×	**42.11**	**2.1**×	**32.63**	**5.2**×	**34.10**	**4.7**×	**36.72**	**4.1**×	**38.59**	**3.4**×	**3.6**×
Memory (GB, ↓)	**14.75**	**0.9**×	**14.53**	**0.9**×	**14.52**	**0.9**×	**15.04**	**1.0**×	**14.95**	**1.0**×	**14.83**	**0.9**×	**14.62**	**0.9**×	**0.9**×

Table 2: Comprehensive benchmark results on Dream-v0-7B and Dream-v0-7B-Instruct (Ye et al. 2025). Each cell presents the accuracy, decoding throughput in tokens per second and peak memory cost in GB with relative efficiency. Best values in bold, suboptimal values underlined.

	Base Model	+dLLM-Cache	+dKV-Cache	+Fast-dLLM	+Sparse-dLLM
- **LLaDA-8B-Instruct**	34.55	33.34	34.72	33.99	34.46
- **LLaDA-1.5**	34.66	33.65	34.68	34.33	34.50
- **Dream-v0-7B-Base**	34.17	34.11	34.16	33.54	34.13
- **Dream-v0-7B-Instruct**	38.62	38.49	38.56	38.00	38.30

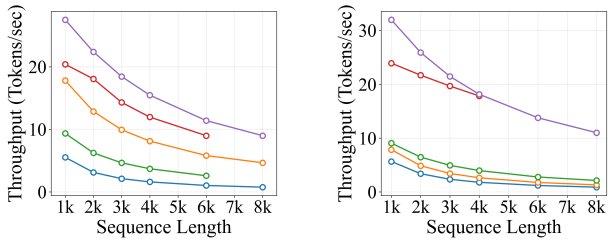
Table 3: Experimental Results on LongBench.

longer sequences.

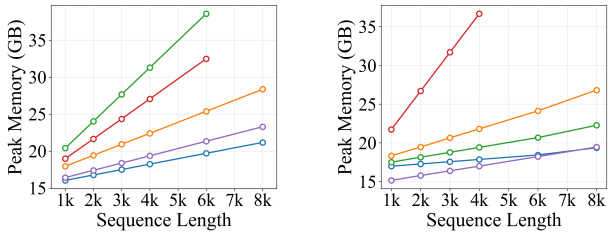
Crucially, these efficiency gains do not compromise performance. Our evaluation on LongBench (Table 3) confirms that Sparse-dLLM preserves the base models' performance with high fidelity. For instance, its 34.46% accuracy on LLaDA-8B-Instruct is nearly identical to the base model's 34.55%. In summary, Sparse-dLLM reconciles the often-conflicting goals of inference efficiency and model performance, establishing it as a robust and practical solution for deploying long-context models.

Ablations and Analysis

N-Step Delayed Cache Updates We systematically examined the effects of the delay step (0-5) for cache updates during decoding a new block on LLaDA-8B-Instruct, as shown in Table 4. The results demonstrate that increasing the delay step leads to progressively lower throughput. Interestingly, accuracy exhibits a non-monotonic pattern, decreasing when transitioning from 1-step to 2-step delay and reaching its peak at 3-step delay. Through a comprehensive analysis of this performance-efficiency trade-off, we



(a) Throughput (LLaDA) (b) Throughput (Dream)



(c) Peak Memory (LLaDA) (d) Peak Memory (Dream)

Figure 5: Efficiency comparison of Vanilla (blue), +dLLM-Cache (orange), +dKV-Cache (green), +Fast-dLLM (red), and +Sparse-dLLM (purple) across varying context lengths. The missing data points indicate an OOM error on the NVIDIA 4090 (48 GB) GPU. We adopt LLaDA-8B-Instruct (LLaDA) and Dream-v0-7B-Instruct (Dream).

Delay Step	0	1	2	3	4	5
Accuracy (%)	86.1	88.47	87.46	**89.49**	88.47	88.14
Throughput (TPS)	**36.89**	36.85	36.26	35.38	34.89	33.78

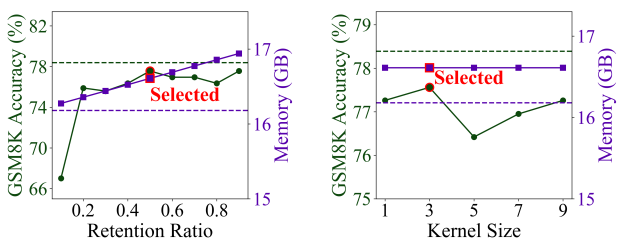
Table 4: Ablation study of delay step on ARC-C benchmark.

	GSM8K	MATH	ARC-C	Avg.
LLaDA-8B-Instruct	78.39	**36.02**	**88.47**	**67.63**
+ Sparse-dLLM	77.56	34.42	**88.47**	66.82
+ prefix-sparse	**78.62**	34.18	83.39	65.40
Dream-v0-7B-Instruct	76.57	39.38	**90.17**	68.71
+ Sparse-dLLM	78.17	**40.48**	88.47	**69.04**
+ prefix-sparse	**78.24**	39.74	89.15	**69.04**

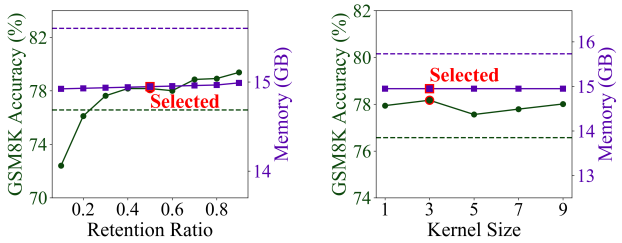
Table 5: Ablation on different sparsity strategies.

conclude that 1-step delay represents the optimal setting, offering near-optimal accuracy while maintaining a near-maximum throughput.

Sparsity Strategy To evaluate the effectiveness of our bidirectional sparsification, we conducted the ablation study presented in Table 5. The results reveal that our Sparse-dLLM, which sparsifies KV states both preceding and succeeding the current block, outperforms the unidirectional prefix-sparse approach. While both strategies enhance performance on the Dream-v0-7B-Instruct model, our approach's advantage is particularly evident on the challenging MATH dataset. Furthermore, although both methods cause a marginal performance drop on the LLaDA-



(a) Retention Ratio (LLaDA) (b) Kernel Size (LLaDA)



(c) Retention Ratio (Dream) (d) Kernel Size (Dream)

Figure 6: Ablation on retention ratio and kernel size. The left vertical axis denotes the accuracy on GSM8K (4-shot), while the right vertical axis represents the peak memory. We adopt LLaDA-8B-Instruct (LLaDA) and Dream-v0-7B-Instruct (Dream).

8B-Instruct model, our approach mitigates this degradation much more effectively. These findings underscore that bidirectional sparsification is a more effective strategy.

Hyperparameters: Retention Ratio and Kernel Size We conducted ablation studies on GSM8K to determine the optimal retention ratio (r) and kernel size (k) for Sparse-dLLM (Figure 6). For the retention ratio, we fix $k = 3$ (Figures 6a and 6c). We observe that increasing r to 0.5 substantially improves accuracy, while further increases yield diminishing returns and linear memory growth. This suggests $r = 0.5$ offers the best trade-off. For the kernel size, we fix $r = 0.5$ (Figures 6b and 6d) and find a clear performance peak at $k = 3$, with accuracy degrading for both smaller and larger kernel sizes. Therefore, we adopt $r = 0.5$ and $k = 3$ for all main experiments.

Conclusion

We introduce Sparse-dLLM, the first training-free method to combine sparse attention with dynamic bidirectional cache eviction for dLLMs. Based on the key insight that attention in dLLMs is both sparse and consistent across decoding steps, our method dynamically evicts unimportant KV cache entries for both prefix and suffix tokens. Our approach achieves three key results: (1) Comparable performance on downstream tasks; (2) State-of-the-art acceleration, with up to 10× higher throughput than vanilla dLLMs; and (3) Optimal memory efficiency, with nearly identical memory costs to vanilla dLLMs.

Acknowledgments

This work was supported by the Shanghai Pilot Program for Basic Research - Fudan University 21TQ1400100 (22TQ018).

References

Berglund, L.; Tong, M.; Kaufmann, M.; Balesni, M.; Stickland, A. C.; Korbak, T.; and Evans, O. 2023. The Reversal Curse: LLMs trained on" A is B" fail to learn" B is A". *arXiv preprint arXiv:2309.12288*.

Bisk, Y.; Zellers, R.; Gao, J.; Choi, Y.; et al. 2020. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, 7432–7439.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. D. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Clark, P.; Cowhey, I.; Etzioni, O.; Khot, T.; Sabharwal, A.; Schoenick, C.; and Tafjord, O. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*.

Cobbe, K.; Kosaraju, V.; Bavarian, M.; Chen, M.; Jun, H.; Kaiser, L.; Plappert, M.; Tworek, J.; Hilton, J.; Nakano, R.; et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Contributors, O. 2023. OpenCompass: A Universal Evaluation Platform for Foundation Models. https://github.com/open-compass/opencompass.

Dziri, N.; Lu, X.; Sclar, M.; Li, X. L.; Jiang, L.; Lin, B. Y.; Welleck, S.; West, P.; Bhagavatula, C.; Le Bras, R.; et al. 2023. Faith and fate: Limits of transformers on compositionality. *Advances in Neural Information Processing Systems*, 36: 70293–70332.

Ge, S.; Zhang, Y.; Liu, L.; Zhang, M.; Han, J.; and Gao, J. 2023. Model tells you what to discard: Adaptive kv cache compression for llms. *arXiv preprint arXiv:2310.01801*.

Gemini. 2025. Gemini Diffusion, our state-of-the-art, experimental text diffusion model.

Hendrycks, D.; Burns, C.; Basart, S.; Zou, A.; Mazeika, M.; Song, D.; and Steinhardt, J. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*.

Hendrycks, D.; Burns, C.; Kadavath, S.; Arora, A.; Basart, S.; Tang, E.; Song, D.; and Steinhardt, J. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Hu, Z.; Meng, J.; Akhauri, Y.; Abdelfattah, M. S.; Seo, J.-s.; Zhang, Z.; and Gupta, U. 2025. Accelerating Diffusion Language Model Inference via Efficient KV Caching and Guided Diffusion. *arXiv preprint arXiv:2505.21467*.

Huang, Z.; Chen, Z.; Wang, Z.; Li, T.; and Qi, G.-J. 2025. Reinforcing the diffusion chain of lateral thought with diffusion language models. *arXiv preprint arXiv:2505.10446*.

Inception. 2025. Introducing Mercury, the world's first commercial-scale diffusion language model.

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. *arXiv preprint arXiv:2407.02490*.

Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C. H.; Gonzalez, J.; Zhang, H.; and Stoica, I. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, 611–626.

Li, Y.; Huang, Y.; Yang, B.; Venkitesh, B.; Locatelli, A.; Ye, H.; Cai, T.; Lewis, P.; and Chen, D. 2025. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 37: 22947–22970.

Liu, Z.; Yang, Y.; Zhang, Y.; Chen, J.; Zou, C.; Wei, Q.; Wang, S.; and Zhang, L. 2025. dLLM-Cache: Accelerating Diffusion Large Language Models with Adaptive Caching.

Ma, X.; Yu, R.; Fang, G.; and Wang, X. 2025. dkv-cache: The cache for diffusion language models. *arXiv preprint arXiv:2505.15781*.

Nie, S.; Zhu, F.; You, Z.; Zhang, X.; Ou, J.; Hu, J.; Zhou, J.; Lin, Y.; Wen, J.-R.; and Li, C. 2025. Large language diffusion models. *arXiv preprint arXiv:2502.09992*.

Rein, D.; Hou, B. L.; Stickland, A. C.; Petty, J.; Pang, R. Y.; Dirani, J.; Michael, J.; and Bowman, S. R. 2024. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*.

Sun, T.; Zhang, X.; He, Z.; Li, P.; Cheng, Q.; Liu, X.; Yan, H.; Shao, Y.; Tang, Q.; Zhang, S.; Zhao, X.; Chen, K.; Zheng, Y.; Zhou, Z.; Li, R.; Zhan, J.; Zhou, Y.; Li, L.; Yang, X.; Wu, L.; Yin, Z.; Huang, X.; Jiang, Y.-G.; and Qiu, X. 2024. MOSS: An Open Conversational Large Language Model. *Machine Intelligence Research*.

Touvron, H.; Lavril, T.; Izacard, G.; Martinet, X.; Lachaux, M.; Lacroix, T.; Rozière, B.; Goyal, N.; Hambro, E.; Azhar, F.; Rodriguez, A.; Joulin, A.; Grave, E.; and Lample, G. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR*, abs/2302.13971.

Wu, C.; Zhang, H.; Xue, S.; Liu, Z.; Diao, S.; Zhu, L.; Luo, P.; Han, S.; and Xie, E. 2025. Fast-dLLM: Training-free Acceleration of Diffusion LLM by Enabling KV Cache and Parallel Decoding. *arXiv preprint arXiv:2505.22618*.

Xiao, G.; Tian, Y.; Chen, B.; Han, S.; and Lewis, M. 2023. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*.

Yang, L.; Tian, Y.; Li, B.; Zhang, X.; Shen, K.; Tong, Y.; and Wang, M. 2025. Mmada: Multimodal large diffusion language models. *arXiv preprint arXiv:2505.15809*.

Ye, J.; Xie, Z.; Zheng, L.; Gao, J.; Wu, Z.; Jiang, X.; Li, Z.; and Kong, L. 2025. Dream 7B.

You, Z.; Nie, S.; Zhang, X.; Hu, J.; Zhou, J.; Lu, Z.; Wen, J.-R.; and Li, C. 2025. LLaDA-V: Large Language Diffusion Models with Visual Instruction Tuning. *arXiv preprint arXiv:2505.16933*.

Yu, R.; Ma, X.; and Wang, X. 2025. Dimple: Discrete Diffusion Multimodal Large Language Model with Parallel Decoding. *arXiv preprint arXiv:2505.16990*.

Zhang, Z.; Sheng, Y.; Zhou, T.; Chen, T.; Zheng, L.; Cai, R.; Song, Z.; Tian, Y.; Ré, C.; Barrett, C.; et al. 2024. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36.

Zhao, S.; Gupta, D.; Zheng, Q.; and Grover, A. 2025. d1: Scaling reasoning in diffusion large language models via reinforcement learning. *arXiv preprint arXiv:2504.12216*.

Zhu, F.; Wang, R.; Nie, S.; Zhang, X.; Wu, C.; Hu, J.; Zhou, J.; Chen, J.; Lin, Y.; Wen, J.-R.; et al. 2025. LLaDA 1.5: Variance-Reduced Preference Optimization for Large Language Diffusion Models. *arXiv preprint arXiv:2505.19223*.