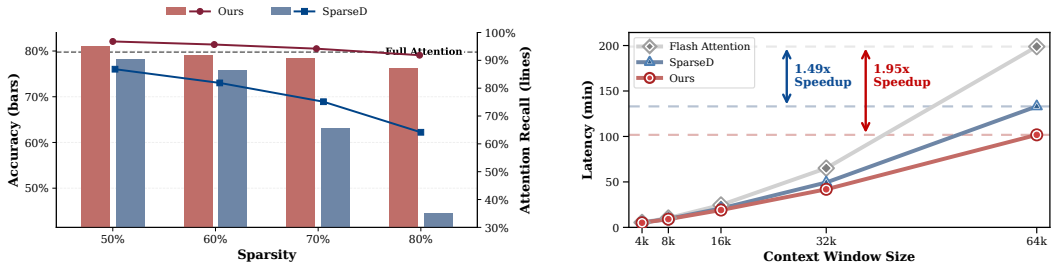
PULSECOL: PERIODICALLY REFRESHED COLUMN-SPARSE ATTENTION FOR ACCELERATING DIFFUSION LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Inference in diffusion large language models (dLLMs) is computationally expensive, as full self-attention must be repeatedly executed at each step of the denoising process without KV cache. Recent sparse attention methods for dLLMs mitigate this cost via block-sparse computation, which is applied only in later iterations when model performance is less sensitive to coarse-grained sparse approximation, but yields limited improvements in computational efficiency and acceleration. This motivates a finer-grained sparsification strategy that can be applied from earlier iterations and leverages reusable sparsity patterns, enabling further efficiency gains. In this work, we introduce PulseCol, a periodically refreshed column-sparse attention method for accelerating diffusion language models. PulseCol replaces coarse block-level sparsity with a finer-grained column-sparse structure, allowing important attention interactions to be retained more precisely while exposing greater sparsity. Built on this column-level formulation, PulseCol further identifies sparse patterns at the early denoising step and reuses them across subsequent iterations, refreshing them only at a small number of intermediate steps to track the evolution of sparse attention patterns during denoising. Experiments show that PulseCol achieves higher sparsity and greater practical speedup than prior sparse attention methods for dLLMs, while maintaining model quality. Enabled by optimized GPU kernels for column-sparse attention, PulseCol delivers up to 1.95× end-to-end speedup over FlashAttention across several context lengths.



(a) GSM8K Accuracy and Attention Recall

(b) Latency and Speedup

Figure 1: PulseCol improves the sparsity-efficiency trade-off of dLLM inference. (a) On GSM8K, PulseCol preserves higher accuracy and oracle per-token top-k attention recall than SparseD under increasing sparsity, indicating that column-level sparsity better retains important attention interactions. (b) On LLaDA-1.5 with 1024 denoising steps, PulseCol achieves lower inference latency than both FlashAttention-based full attention and SparseD, reaching a $1.95\times$ speedup at a 64K context window.

1 INTRODUCTION

Diffusion language models have recently attracted growing interest as a complementary direction to autoregressive language modeling (Austin et al., 2021; Li et al., 2022; Zheng et al., 2023; Lou et al., 2023; Sahoo et al., 2024; Ou et al., 2024; Gong et al., 2024; Nie et al., 2025; Ye et al., 2025).

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.

By iteratively refining a sequence with a bidirectional context, dLLMs offer a flexible generation paradigm beyond the strict left-to-right decoding (Nie et al., 2025; Ye et al., 2025). However, this flexibility comes with a substantial inference cost, since full attention must be recomputed at each denoising step (Ma et al., 2025; Wu et al., 2025; Liu et al., 2025; Wang et al., 2025b). This attention cost accumulates across repeated denoising steps, hindering efficient dLLM inference.

Given this accumulated attention cost, exploiting the sparsity in attention is a natural way to improve the dLLM inference efficiency. However, sparse attention methods developed for autoregressive language models do not transfer directly to dLLMs because their inference processes differ fundamentally (Child et al., 2019; Zaheer et al., 2020; Xiao et al., 2023; Zhang et al., 2023; Lai et al., 2025; Wang et al., 2025b; Xi et al., 2026). Autoregressive models generate tokens by extending a prefix over time and reusing a persistent KV (Kwon et al., 2023; Xiao et al., 2023; Zhang et al., 2023), whereas dLLMs iteratively update a sequence-level denoising state (Nie et al., 2025; Ye et al., 2025; Ma et al., 2025; Wu et al., 2025). As a result, sparse attention in dLLMs cannot be determined solely by static token positions or a single-step importance estimate, since useful attention connections may change as denoising progresses (Song et al., 2026; Wang et al., 2025b; Xi et al., 2026). These differences call for sparse attention designs that account for evolving attention patterns during denoising.

Recent studies on dLLM inference have further observed that attention patterns exhibit noticeable similarity across adjacent denoising steps (Ma et al., 2025; Liu et al., 2025; Song et al., 2026; Wang et al., 2025b; Xi et al., 2026). This observation suggests that the sparse attention structure need not be recomputed from scratch at every iteration, motivating a line of methods that accelerate dLLM inference by reusing sparsity patterns over the denoising process. Despite their effectiveness, these methods typically apply sparsification only in the later stages of inference. The underlying assumption is that early denoising steps are highly influential to the final generation quality and therefore should preserve full attention (Wang et al., 2025b). This design choice reveals a key unresolved question. Why is early-stage attention sparsification in dLLMs so fragile?

In this work, we investigate attention patterns in early denoising iterations. We observe that attention in the early stage of inference is not truly dense. Instead, high-scoring attention is consistently concentrated on a small number of key columns, revealing a pronounced column-wise sparsity pattern. This finding provides a possible explanation for the fragility of existing sparse dLLM inference methods in early steps. Their block-level approximations impose a coarse sparsity granularity that is misaligned with the intrinsic column-wise structure of attention, and may therefore discard important attention connections during early denoising.

Motivated by this observation, we propose PulseCol, a periodically refreshed column-sparse attention method for efficient dLLM inference. Instead of approximating attention with coarse block-level sparsity (Wang et al., 2025b; Xi et al., 2026), PulseCol directly targets the column-wise sparsity structure prominent in early denoising steps. By replacing block sparsity with column sparsity, PulseCol better approximates the underlying attention patterns, enabling sparse attention to be introduced more safely from earlier denoising stages. Since attention patterns still evolve during denoising, PulseCol reuses column-sparse indices across nearby steps while periodically refreshing them to adapt to this evolution. This design enables early-stage sparsity reuse without assuming fixed sparsity throughout inference, thereby expanding the acceleration window of dLLM inference.

Experiments show that PulseCol achieves a favorable trade-off between generation quality and inference efficiency. Under the same sparsity budget, PulseCol outperforms prior sparse-attention baselines on average. At 80% sparsity, it remains only slightly behind full attention while maintaining a clear advantage over existing sparse methods. With our optimized GPU kernel for column-sparse attention, PulseCol further translates its algorithmic sparsity into practical inference speedups, achieving up to 1.95× speedup over FlashAttention (Dao, 2023) across different context lengths.

2 Related Works

2.1 Diffusion Language Models

Diffusion language models have recently been explored as a complementary approach to autoregressive generation (Nie et al., 2025; Zhu et al., 2025; Bie et al., 2025; Ye et al., 2025). Recent work such as LLaDA (Nie et al., 2025) has shown that this paradigm can scale to large models under standard

pretraining and instruction tuning, while supporting parallel refinement over an entire sequence with competitive performance. These strengths have made dLLMs an increasingly attractive framework for language generation. At the same time, their iterative denoising process repeatedly invokes bidirectional full attention over multiple steps, making inference efficiency a central challenge (Wu et al., 2025; Wang et al., 2025b).

Existing efforts to accelerate dLLM inference largely fall into two categories. The first exploits KV caching and cross-step reuse. Methods such as Fast-dLLM (Wu et al., 2025), FlashDLM (Hu et al., 2025), and dLLM-Cache (Liu et al., 2025) reduce redundant computation across neighboring denoising steps through approximate caching, selective updates, or more effective reuse of intermediate states. The second focuses on sampling and decoding policies. These approaches improve inference efficiency through confidence- or entropy-aware scheduling (Wu et al., 2025; Ben-Hamu et al., 2025), pipelined generation (Wang et al., 2025a), or distillation-based methods designed for parallel decoding (Chen et al., 2025). While effective, these methods primarily reduce the number of effective iterations or reuse computation across steps, rather than addressing the cost of attention computation within each denoising step.

2.2 Sparse Attention for dLLMs

Sparse attention has long been a common approach for reducing attention cost and has been studied extensively in autoregressive models. By comparison, its application to dLLMs remains much less explored (Wang et al., 2025b; Xi et al., 2026). SparseD is one of the first works to explore sparse attention for dLLMs. Its key idea is to reuse sparse attention patterns across denoising steps, while preserving full attention in early iterations and adopting block-wise sparsity to balance stability and efficiency. Beyond sparse attention methods themselves, recent analysis of dLLM attention dynamics suggests that attention in dLLMs exhibits distinctive dynamic properties. In particular, Rulli et al. (2025) shows that sink locations in dLLMs shift over the denoising process and are often associated with semantic or structural tokens, in contrast to the more stable sink patterns commonly observed in autoregressive models. Together, these studies suggest that attention in dLLMs exhibits both reusable structure across steps and dynamic variation over the course of refinement, which makes the timing, update frequency, and granularity of sparsification especially important.

Our work studies how to sparsify attention effectively during iterative dLLM inference. Unlike prior methods that introduce sparsity in later iterations and rely on block-level constraints, we exploit reusable attention structure earlier in the denoising process using finer-grained sparse patterns.

3 Method

3.1 Preliminaries and Problem Setup

Discrete Diffusion Language Models. dLLMs formulate text generation as an iterative denoising process over discrete tokens. Given a prompt $c = (c_1, \ldots, c_m)$ and a target generation length N, the response is initialized as fully masked, forming $x_T = [c_1, \ldots, c_m, [\mathrm{M}], \ldots, [\mathrm{M}]]$, where $[\mathrm{M}]$ denotes the mask token. A pretrained denoising model then gradually transforms this fully masked response into a complete sequence through T reverse denoising steps. At step t, let $\mathcal{M}_t = \{i : x_t^i = [\mathrm{M}]\}$ be the set of masked positions. The model predicts token distributions $p_\theta(x_0^i \mid x_t)$ for all $i \in \mathcal{M}_t$ in parallel, and a decoding policy selects a subset $\mathcal{U}_t \subseteq \mathcal{M}_t$ to unmask, often according to a generation schedule and token confidence. For example, the confidence of position i can be measured as $\max_v p_\theta(v \mid x_t)$, so high-confidence positions are committed earlier while the remaining positions stay masked and are refined in later steps. The process terminates when all response positions are unmasked. Unlike autoregressive language models, which produce one token at a time, dLLMs can refine multiple positions within the same denoising step. This parallel refinement makes dLLMs attractive for efficient generation. However, since the denoising network must be repeatedly evaluated over the evolving sequence across multiple steps, the overall inference cost remains substantial, especially when the sequence is long or the number of denoising steps is large.

Sparse Attention for dLLM Inference. At denoising step t, consider the attention computation at model layer ℓ and head h. Given hidden states of length $n = m + N$, the attention logits are

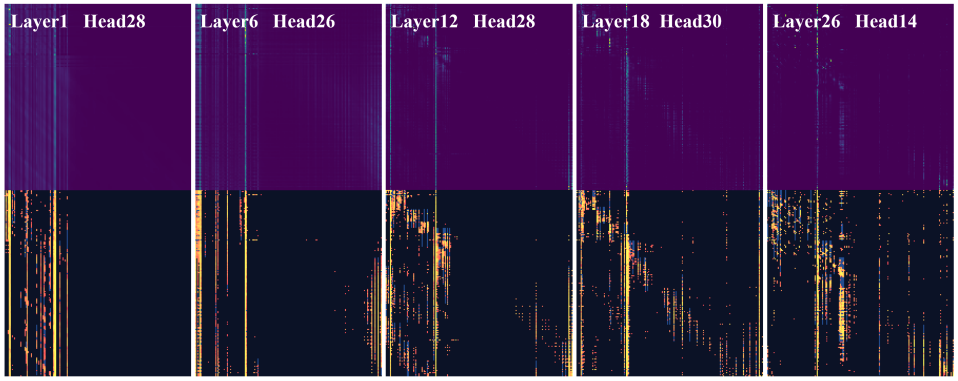


Figure 2: Early denoising attention in LLaDA-1.5 exhibits column sparsity. We visualize representative attention heads at an early denoising step, step 4 of 128. The top row shows attention heatmaps, with brighter colors indicating larger attention scores. The bottom row shows per-token top-k attention positions, where brighter orange denotes higher-ranked positions and the blue background marks the columns selected by our group-wise column-sparse pattern. Important attention mass concentrates on a small set of columns, motivating column-level sparsity in early denoising.

computed as

$$Z_t^{\ell,h} = \frac{Q_t^{\ell,h}(K_t^{\ell,h})^\top}{\sqrt{d_h}}, \tag{1}$$

where $Q_t^{\ell,h}, K_t^{\ell,h}, V_t^{\ell,h} \in \mathbb{R}^{n\times d_h}$. Standard full attention then produces $A_t^{\ell,h} = \mathrm{softmax}(Z_t^{\ell,h})V_t^{\ell,h}$. Since dLLMs repeatedly perform full-sequence bidirectional attention over T denoising steps, the total attention cost scales as $O(TLHn^2d_h)$, where L is the number of layers and H is the number of heads. This repeated quadratic computation makes attention a major bottleneck in dLLM inference.

Sparse attention reduces this cost by restricting the set of query-key interactions. For each step, layer, and head, let $S_t^{\ell,h} \in \{0,1\}^{n\times n}$ be a sparse attention mask, where $S_t^{\ell,h}(i,j) = 1$ indicates that query position i can attend to key position j. The sparse attention output is computed as

$$\widetilde{A}_t^{\ell,h} = \mathrm{softmax}\left(Z_t^{\ell,h} + M(S_t^{\ell,h})\right) V_t^{\ell,h}, \tag{2}$$

where $M(S_t^{\ell,h})_{ij} = 0$ if $S_t^{\ell,h}(i,j) = 1$, and $M(S_t^{\ell,h})_{ij} = -\infty$ otherwise.

We characterize the sparsity level by the number of retained query-key interactions in each attention head. Specifically, for a target sparsity ρ, each sparse mask is constrained by

$$\|S_t^{\ell,h}\|_0 \le (1-\rho)n^2, \quad \forall\, t,\ell,h. \tag{3}$$

Under a fixed computation budget, an effective sparse attention pattern should retain query-key interactions that are important for the current denoising prediction while removing those with limited contribution. A common way to impose such sparsity is through block-sparse attention, which reduces the selection space by grouping tokens into blocks. Suppose the sequence is partitioned into blocks $\{\mathcal{B}_1,\ldots,\mathcal{B}_B\}$, and let $b(i)$ denote the block index of token i. The token-level sparse mask $S_t^{\ell,h} \in \{0,1\}^{n\times n}$ is then induced by a block-level binary matrix $G_t^{\ell,h} \in \{0,1\}^{B\times B}$, where

$$S_t^{\ell,h}(i,j) = G_t^{\ell,h}(b(i),b(j)). \tag{4}$$

This design forces all token pairs within the same block pair to share a sparsity decision, which may retain low-importance interactions while discarding isolated important ones, leading to information loss in early denoising steps.

3.2 Motivation

Prior sparse attention work on dLLMs adopts a block-sparse design and treats early denoising steps as more sensitive to sparse approximation, therefore keeping full attention in these steps (Wang et al.,

2025b). This design is motivated by the empirical observation that directly applying the block-sparse approximation to early denoising steps can lead to noticeable performance degradation.

However, this degradation does not necessarily imply that early attention lacks exploitable sparse structure. It may instead reflect a mismatch between early-stage attention patterns and the block-sparse structure used in prior work. We therefore revisit early-stage attention maps to examine whether they contain structures that can support a more suitable sparsification strategy.

Our analysis shows that early denoising attention already exhibits a clear column-sparse pattern. As illustrated in Figure 2, high-scoring attention interactions involve only a small subset of keys, which are repeatedly attended to by multiple query positions, while many other keys contribute little to the attention output. This suggests that important interactions in early dLLM attention are better characterized by fine-grained sparsity along the key dimension than by coarse block-shaped regions. Appendix A provides further visualizations and quantitative analyses across layers and heads.

This observation provides a new perspective on why prior block-sparse work avoids sparse attention in early denoising steps. Early attention does contain exploitable sparsity, but block-level approximation is too coarse to match its fine-grained key-wise structure. Since important interactions are concentrated around a small subset of keys, column-sparse patterns provide a better aligned sparsity form and can enable sparse computation in earlier denoising steps.

After identifying the exploitable column-sparse structure in early attention, the next question is how to maintain such sparse structure efficiently throughout iterative denoising. Prior work on efficient dLLM inference has observed similar attention patterns across adjacent denoising iterations, providing the basis for reusing sparse indices across steps. However, for early denoising, directly fixing the initially constructed sparse indices for a long interval may still be unstable. As token predictions and attention patterns change across denoising steps, the important keys selected at the beginning may no longer align with those receiving high attention scores in later iterations. Therefore, PulseCol introduces a low-frequency periodic refresh mechanism in the early inference stage, allowing sparse indices to be updated when the attention structure is still rapidly evolving.

Based on these observations, we propose PulseCol, a periodically refreshed column-sparse attention method for accelerating dLLM inference. PulseCol follows three design choices. First, it adopts column-level sparsity to match early denoising attention, where important interactions involve only a small subset of keys. Second, it enables sparse computation in earlier denoising steps by selecting important keys rather than coarse blocks, reducing the granularity mismatch of block-sparse approximation. Third, it balances reuse and adaptation by reusing selected keys across nearby denoising steps and periodically refreshing them in the early stage as attention patterns change.

As illustrated in Figure 3, PulseCol is organized as a three-stage pipeline that follows the design choices above. In the Initial Sparse Pattern stage, PulseCol constructs group-wise column-sparse indices by selecting important keys from the first denoising step. In the Periodic Sparsity Refresh stage, it keeps applying sparse attention in early denoising steps while refreshing the sparse indices at a small number of intermediate steps, allowing the sparse pattern to remain aligned with changing attention patterns. In the Persistent Sparsity Reuse stage, PulseCol fixes the sparse indices after the early refresh period and reuses them for the remaining steps to avoid repeated pattern construction.

3.3 PULSECOL

Group-wise Column-Sparse Pattern Estimation. At each refresh step τ_r, PulseCol computes full attention to collect $P^{(\tau_r)} \in \mathbb{R}^{n\times n}$ and estimate a group-wise column-sparse pattern. We partition query positions into groups $\{\mathcal{G}_1, \ldots, \mathcal{G}_U\}$, where queries in the same group share one set of selected keys. For each query group $\mathcal{G}_u$, PulseCol computes a key importance score by aggregating attention weights within the group as

$$s_{u,j}^{(\tau_r)} = \frac{1}{|\mathcal{G}_u|} \sum_{i\in\mathcal{G}_u} P_{ij}^{(\tau_r)}, \qquad j = 1, \ldots, n. \tag{5}$$

The score $s_{u,j}^{(\tau_r)}$ measures how strongly the j-th key is attended to by queries in $\mathcal{G}_u$ on average. PulseCol then selects the top-k keys for each group,

$$C_u^{(r)} = \mathrm{TopK}\left(s_u^{(\tau_r)}, k\right). \tag{6}$$

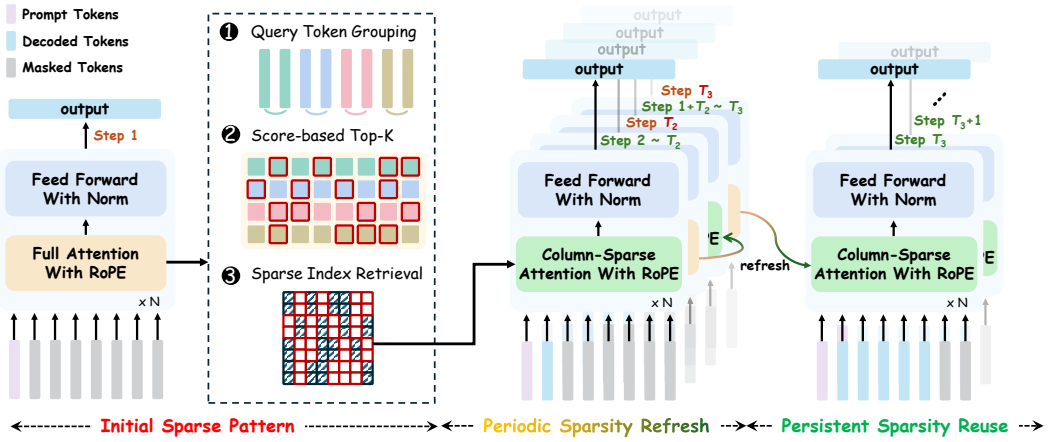


Figure 3: Overview of PulseCol. By constructing, refreshing, and reusing column-sparse indices, PulseCol enables early sparse attention while limiting pattern-construction cost.

The resulting sparse mask is shared within each query group, with $S_{ij}^{(r)} = \mathbf{1}\{j \in C_u^{(r)}\}$ for $i \in \mathcal{G}_u$. This selection is equivalent to solving a group-level column selection objective formulated as

$$C_u^{(r)} \in \arg\max_{\substack{C \subseteq [n] \\ |C|=k}} \sum_{i \in \mathcal{G}_u} \sum_{j \in C} P_{ij}^{(\tau_r)}. \tag{7}$$

The objective selects the keys that preserve the largest aggregate attention scores for each query group. This group-wise design provides a middle ground between per-query key selection and head-level global sharing. Per-query selection can better match individual attention distributions but produces highly irregular indices. Head-level sharing is more regular but may ignore differences across query regions. By sharing selected keys only within local query groups, PulseCol preserves region-specific important interactions while keeping the sparse pattern structured enough for efficient reuse.

Compared with block-sparse patterns, group-wise column sparsity selects keys directly rather than retaining all token pairs inside selected blocks. It therefore avoids introducing many low-contribution interactions solely due to block boundaries, while reducing the chance of discarding isolated important keys. This makes the estimated pattern better aligned with the fine-grained key-wise structure observed in early denoising attention.

Periodic Sparse Pattern Refresh and Persistent Reuse. After obtaining the column sets $C_u^{(r)}$ for all query groups, PulseCol reuses them across subsequent denoising steps. We use a simple refresh schedule controlled by two hyperparameters. The first is a refresh-window ratio $\eta \in (0, 1]$, which specifies the fraction of inference steps where sparse patterns can be refreshed. The second is a refresh budget R, which specifies how many refreshes are performed within this interval. Given a total of T denoising steps, we define the refresh window length as $T_{\text{win}} = \lfloor \eta T \rfloor$. We distribute the R refreshes uniformly over the first T_{win} steps, yielding the refresh set $\mathcal{T}_{\text{ref}} = \{\tau_1, \tau_2, \ldots, \tau_R\}$. When $R = 1$, we set $\tau_1 = 1$. For $R \geq 2$,

$$\tau_r = 1 + \left\lfloor \frac{(r-1)(T_{\text{win}} - 1)}{R - 1} \right\rfloor, \qquad r = 1, \ldots, R. \tag{8}$$

Under this schedule, sparse patterns are refreshed only during the early inference stage and are reused between consecutive refresh steps. When $t > T_{\text{win}}$, PulseCol stops refreshing the pattern and keeps using the column sets obtained at the last refresh. This design allows the sparse indices to adapt during early denoising while avoiding repeated pattern construction in later steps.

3.4 Column-Sparse Attention Kernel

We implement a column-sparse attention kernel that computes attention over a given set of sparse column indices. Following FlashAttention, the kernel operates on tiles, keeps intermediate softmax

statistics in on-chip memory, and avoids materializing the full attention matrix in HBM. Unlike dense kernels, it traverses only the indexed key-value columns, so each query block attends to the positions specified by its sparse index list. Figure 4 shows the kernel workflow.

Given $Q, K, V \in \mathbb{R}^{n \times d_h}$, we partition the query sequence into n_q blocks of size B_M. The i-th query block is denoted by $Q_i \in \mathbb{R}^{B_M \times d_h}$. The column sparse pattern is represented by an index tensor $S \in \mathbb{N}^{n_q \times n_s}$, where $S[i,:]$ stores the n_s key-value position indices selected for query block i. Thus, rather than attending to the entire key value sequence, the computation for Q_i is restricted to the subset of columns indexed by $S[i,:]$.

The kernel schedules computation at the granularity of query blocks. For each query block i, it loads Q_i from HBM into SRAM, initializes the row-wise online softmax statistics and output accumulator, and then traverses the sparse column indices in tiles of size B_N. At each step, the kernel reads a contiguous tile from $S[i,:]$, gathers the corresponding rows of K and V, computes attention scores with the current query tile, and updates the online softmax state and output accumulator in SRAM without materializing the full attention matrix. After all n_s selected key-value columns have been processed, the accumulated output is normalized with the maintained softmax statistics and written back to HBM. Repeating this procedure across query blocks preserves the memory efficiency of FlashAttention while supporting externally specified sparse patterns, with both computation and memory traffic scaling with n_s rather than the full sequence length.

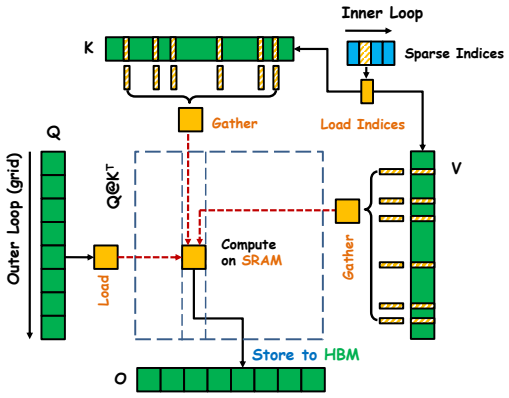


Figure 4: Workflow of the column-sparse attention kernel. Each query block attends only to indexed key-value tiles, updates online softmax statistics in SRAM, and writes the normalized output back to HBM without materializing the full attention matrix.

4 Experiments

4.1 Experimental Setup

Baselines and Datasets. We evaluate PulseCol on LLaDA-1.5 (Zhu et al., 2025) and Dream (Ye et al., 2025), comparing against full attention, Sliding Window, StreamingLLM (Xiao et al., 2023), and SparseD (Wang et al., 2025b). We evaluate mathematical reasoning on GSM8K (Cobbe et al., 2021) and MATH-500 (Lightman et al., 2023), code generation on HumanEval (Chen et al., 2021), and long-context understanding on RULER (Hsieh et al., 2024). We evaluate GSM8K in a 4-shot setting and HumanEval and RULER in 0-shot settings.

Implementation Details. We follow SparseD's baseline evaluation settings and use FlashAttention for full attention. Sliding Window and StreamingLLM use windows of 256 tokens for short-context tasks and 2048/4096 tokens for RULER-4K/8K; StreamingLLM also retains the first 10% of key tokens as sinks. SparseD uses `block_size` $= 32$ for short-context tasks and 128 for RULER, with $\rho \in \{50\%, 80\%\}$ and `skip` $= 20\%$. PulseCol follows SparseD's task-specific group sizes, using the same query-group configuration for accuracy and end-to-end latency measurements on RULER, where long contexts make attention a major bottleneck. PulseCol uses $\eta = 30\%$ and refresh budgets $R = 16/8$ for LLaDA-1.5/Dream, respectively.

4.2 Main Result

Accuracy. Table 1 reports the main results on LLaDA-1.5 and Dream. Overall, PulseCol maintains competitive performance at high sparsity. On LLaDA-1.5, PulseCol achieves an average score of 60.94 at 80% sparsity, only 1.88 points below the full-attention baseline while outperforming SparseD at the same sparsity by 7.12 points. Reducing sparsity to 50% increases the average score to 62.34, narrowing the gap to full attention to 0.48 points and achieving performance comparable to SparseD.

Table 1: Main results across benchmarks. Sparsity denotes attention sparsity.

Model	Sparsity	Method	GSM8K	MATH-500	HE	RULER-4K	RULER-8K	Avg.
LLaDA-1.5	0%	Full Attention	79.76	43.80	39.63	90.69	60.22	62.82
	–	Sliding Window	54.28	29.20	28.05	43.35	32.65	37.51
	–	StreamingLLM	58.98	35.80	37.20	45.02	38.38	43.08
	50%	SparseD	78.24	41.20	39.63	90.93	61.99	62.40
		Ours	81.05	41.80	38.41	90.36	60.06	62.34
	80%	SparseD	44.43	33.80	38.41	90.56	61.88	53.82
		Ours	76.19	38.40	38.41	91.72	59.98	60.94
Dream	0%	Full Attention	76.65	51.40	56.71	89.90	70.98	69.13
	–	Sliding Window	70.13	28.80	45.73	46.74	31.36	44.55
	–	StreamingLLM	69.98	47.80	55.49	50.52	36.91	52.14
	50%	SparseD	72.55	49.60	54.87	89.83	66.16	66.60
		Ours	77.79	52.40	57.93	89.82	70.77	69.74
	80%	SparseD	38.44	48.20	43.29	81.55	61.76	54.65
		Ours	74.07	47.00	57.32	89.40	70.53	67.66

The advantage of PulseCol is more pronounced on Dream. At 80% sparsity, SparseD's average score drops to 54.65, whereas PulseCol achieves 67.66, outperforming SparseD by 13.01 points and remaining only 1.47 points below the full-attention baseline. These results indicate that PulseCol preserves model performance more effectively than the block-sparse baseline at high sparsity. At 50% sparsity, PulseCol further improves its average score to 69.74, exceeding full attention and SparseD by 0.61 and 3.14 points, respectively.

Latency. We evaluate the standalone efficiency of the column-sparse attention kernel. Unless otherwise stated, all latency measurements are conducted on an NVIDIA RTX PRO 6000 GPU. Figure 5 shows the speedup over FlashAttention with the query group size fixed to $B_M = 128$. The column-sparse kernel achieves consistent speedups across all tested settings, and the speedup generally increases with context length. For example, at a sparsity of 90%, the speedup improves from 7.21× to 10.54× as the context length increases from 4K to 128K tokens. Results with other query group sizes are reported in Appendix 10 and follow the same trend. These results demonstrate practical kernel-level acceleration from column sparsity.

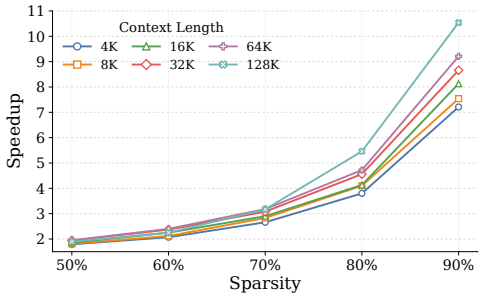
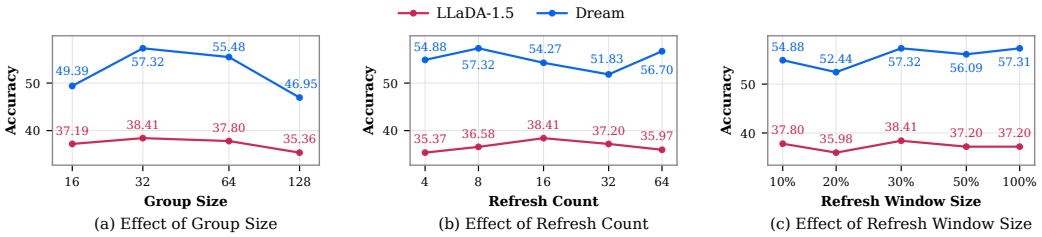


Figure 5: Speedup of our column-sparse kernel over FlashAttention under different sparsity levels and context lengths.

We further evaluate end-to-end inference latency, including the cost of the dedicated score-collection kernel used for sparse-index construction. Figure 1(b) reports results on LLaDA-1.5 with 1024 denoising steps. Compared with the FlashAttention baseline, our method achieves up to 1.95× speedup at 64K context, with larger gains at longer contexts. The same trend is observed on Dream, with full results in Appendix D.3. Our method also outperforms SparseD, reducing latency by 23.5% on LLaDA-1.5 at 64K context. This is because SparseD mainly applies block sparsity in late denoising stages, whereas our method enables sparse computation earlier and further benefits from the column-sparse kernel. Complete latency results across denoising steps are reported in Appendix D.2.

4.3 Ablation Study

Effectiveness of Each Component. We ablate the sparsity pattern and refresh strategy in Table 2. Replacing column sparsity with block sparsity causes performance drops on both models, with an average degradation of 8.16 points across the two benchmarks. This indicates that block sparsity fails



(a) Effect of Group Size (b) Effect of Refresh Count (c) Effect of Refresh Window Size

Figure 6: Hyperparameter analysis on HumanEval. We study the effects of query group size, sparse-index refresh count, and refresh window size on LLaDA-1.5 and Dream.

to capture critical early-stage attention patterns that are better represented by column-wise selection. For a fair comparison, Skip and Periodic use the same number of full-attention steps, differing only in how these steps are distributed over the denoising process. Replacing periodic refresh with SparseD's skipping sparse also degrades performance on both models, with an average drop of 5.00 points across the two benchmarks. This suggests that sparse patterns should be updated during denoising, as a fixed pattern can become misaligned with the evolving attention distribution. Overall, these results show that column sparsity and periodic refresh are both key components of our method, and their combination achieves the best performance.

Hyperparameter Analysis. We study the effect of query group size in Figure 6(a). Performance peaks at a group size of 32, while larger groups lead to a clear drop. This suggests that the group size needs to balance indexing overhead and query-specific sparsity. Larger groups reduce indexing overhead by sharing column indices across more queries, but the shared indices become less aligned with individual attention patterns, weakening the approximation quality.

Table 2: Ablation of sparsity patterns and scheduling strategies. *Periodic* denotes periodic refresh, while *Skip* follows the skipping sparse strategy of SparseD.

		LLaDA-1.5		Dream	
Pattern	Strategy	GSM8k	HE	GSM8k	HE
Column	Periodic	**76.19**	**38.41**	**74.07**	**57.32**
Block	Periodic	58.00	33.54	72.40	49.39
Column	Skip	69.14	34.15	70.28	52.44

Figure 6(b) shows the effect of the number of sparse-index refreshes. The performance is non-monotonic on both models. LLaDA-1.5 peaks at 16 refreshes, while Dream peaks at 8 refreshes, and further increasing the refresh count does not consistently improve performance. This reflects a trade-off in refresh frequency. Too few refreshes make the sparse indices stale, while overly frequent refreshes can make the discrete column selection sensitive to transient changes in attention scores, leading to unstable sparse supports.

We further analyze the effect of the sparse-index refresh window, as shown in Figure 6(c). Refreshing sparse indices in the early denoising stage is usually sufficient and gives the best trade-off. This suggests that index updates are most useful early in denoising, when token distributions may change more substantially. Later iterations appear to mainly refine already-formed predictions, making additional refreshes less beneficial. This supports our choice of limiting refreshes to the early stage.

5 Conclusion

In this work, we introduced PulseCol, a periodically refreshed column-sparse attention method for accelerating dLLM inference. By using column-level sparsity, PulseCol retains important attention interactions more precisely than block-sparse alternatives and enables sparse computation from earlier denoising steps. By reusing sparse indices across nearby iterations and refreshing them only in the early stage, it reduces pattern construction overhead while tracking changes in attention. With optimized column-sparse kernels, PulseCol achieves practical end-to-end speedups over FlashAttention and prior dLLM sparse attention methods while maintaining competitive model quality. Our results show that combining fine-grained sparsity with temporal reuse is an effective way to reduce the inference cost of diffusion language models.

AI USE STATEMENT

We used generative AI tools to polish the language of the manuscript and improve its clarity and readability.

REFERENCES

Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. Structured denoising diffusion models in discrete state-spaces. *Advances in neural information processing systems*, 34:17981–17993, 2021.

Heli Ben-Hamu, Itai Gat, Daniel Severo, Niklas Nolte, and Brian Karrer. Accelerated sampling from masked diffusion models via entropy bounded unmasking. *arXiv preprint arXiv:2505.24857*, 2025.

Tiwei Bie, Maosong Cao, Kun Chen, Lun Du, Mingliang Gong, Zhuochen Gong, Yanmei Gu, Jiaqi Hu, Zenan Huang, Zhenzhong Lan, et al. Llada2. 0: Scaling up diffusion language models to 100b. *arXiv preprint arXiv:2512.15745*, 2025.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Zigeng Chen, Gongfan Fang, Xinyin Ma, Ruonan Yu, and Xinchao Wang. dparallel: Learnable parallel decoding for dllms. *arXiv preprint arXiv:2509.26488*, 2025.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

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

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Shansan Gong, Shivam Agarwal, Yizhe Zhang, Jiacheng Ye, Lin Zheng, Mukai Li, Chenxin An, Peilin Zhao, Wei Bi, Jiawei Han, et al. Scaling diffusion language models via adaptation from autoregressive models. *arXiv preprint arXiv:2410.17891*, 2024.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024.

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. *arXiv preprint arXiv:2505.21467*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Xunhao Lai, Jianqiao Lu, Yao Luo, Yiyuan Ma, and Xun Zhou. Flexprefill: A context-aware sparse attention mechanism for efficient long-sequence inference. *arXiv preprint arXiv:2502.20766*, 2025.

Xiang Li, John Thickstun, Ishaan Gulrajani, Percy S Liang, and Tatsunori B Hashimoto. Diffusion-lm improves controllable text generation. *Advances in neural information processing systems*, 35: 4328–4343, 2022.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

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.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. *arXiv preprint arXiv:2310.16834*, 2023.

Xinyin Ma, Runpeng Yu, Gongfan Fang, and Xinchao Wang. dkv-cache: The cache for diffusion language models. *arXiv preprint arXiv:2505.15781*, 2025.

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*, 2025.

Jingyang Ou, Shen Nie, Kaiwen Xue, Fengqi Zhu, Jiacheng Sun, Zhenguo Li, and Chongxuan Li. Your absorbing discrete diffusion secretly models the conditional distributions of clean data. *arXiv preprint arXiv:2406.03736*, 2024.

Maximo Eduardo Rulli, Simone Petruzzi, Edoardo Michielon, Fabrizio Silvestri, Simone Scardapane, and Alessio Devoto. Attention sinks in diffusion language models. *arXiv preprint arXiv:2510.15731*, 2025.

Subham S Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. *Advances in Neural Information Processing Systems*, 37:130136–130184, 2024.

Yuerong Song, Xiaoran Liu, Ruixiao Li, Zhigeng Liu, Zengfeng Huang, Qipeng Guo, Ziwei He, and Xipeng Qiu. Sparse-dllm: Accelerating diffusion llms with dynamic cache eviction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 33038–33046, 2026.

Xu Wang, Chenkai Xu, Yijie Jin, Jiachun Jin, Hao Zhang, and Zhijie Deng. Diffusion llms can do faster-than-ar inference via discrete diffusion forcing. *arXiv preprint arXiv:2508.09192*, 2025a.

Zeqing Wang, Gongfan Fang, Xinyin Ma, Xingyi Yang, and Xinchao Wang. Sparsed: Sparse attention for diffusion language models. *arXiv preprint arXiv:2509.24014*, 2025b.

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.

Haocheng Xi, Harman Singh, Yuezhou Hu, Coleman Hooper, Rishabh Tiwari, Aditya Tomar, Minjae Lee, Wonjun Kang, Michael Mahoney, Chenfeng Xu, et al. Losa: Locality aware sparse attention for block-wise diffusion language models. *arXiv preprint arXiv:2604.12056*, 2026.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

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*, 2025.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36:34661–34710, 2023.

Lin Zheng, Jianbo Yuan, Lei Yu, and Lingpeng Kong. A reparameterized discrete diffusion model for text generation. *arXiv preprint arXiv:2302.05737*, 2023.

Fengqi Zhu, Rongzhen Wang, Shen Nie, Xiaolu Zhang, Chunwei Wu, Jun Hu, Jun Zhou, Jianfei Chen, Yankai Lin, Ji-Rong Wen, et al. Llada 1.5: Variance-reduced preference optimization for large language diffusion models. *arXiv preprint arXiv:2505.19223*, 2025.

A Additional Analysis of Column-Sparse Patterns

A.1 Additional Visualizations of Column-Sparse Patterns

Figure 7 provides additional attention visualizations from LLaDA-1.5. For each layer, we use a representative attention head to show the attention scores and the corresponding per-token top-k attention positions. These visualizations follow the same format as Figure 2: the heatmaps show attention scores, the orange marks indicate the ranked top-k positions, and the blue background denotes the columns selected by our group-wise column-sparse pattern. The additional examples show that high-attention positions are concentrated on a small subset of columns across different layers, further supporting the column-sparse pattern used by PulseCol.

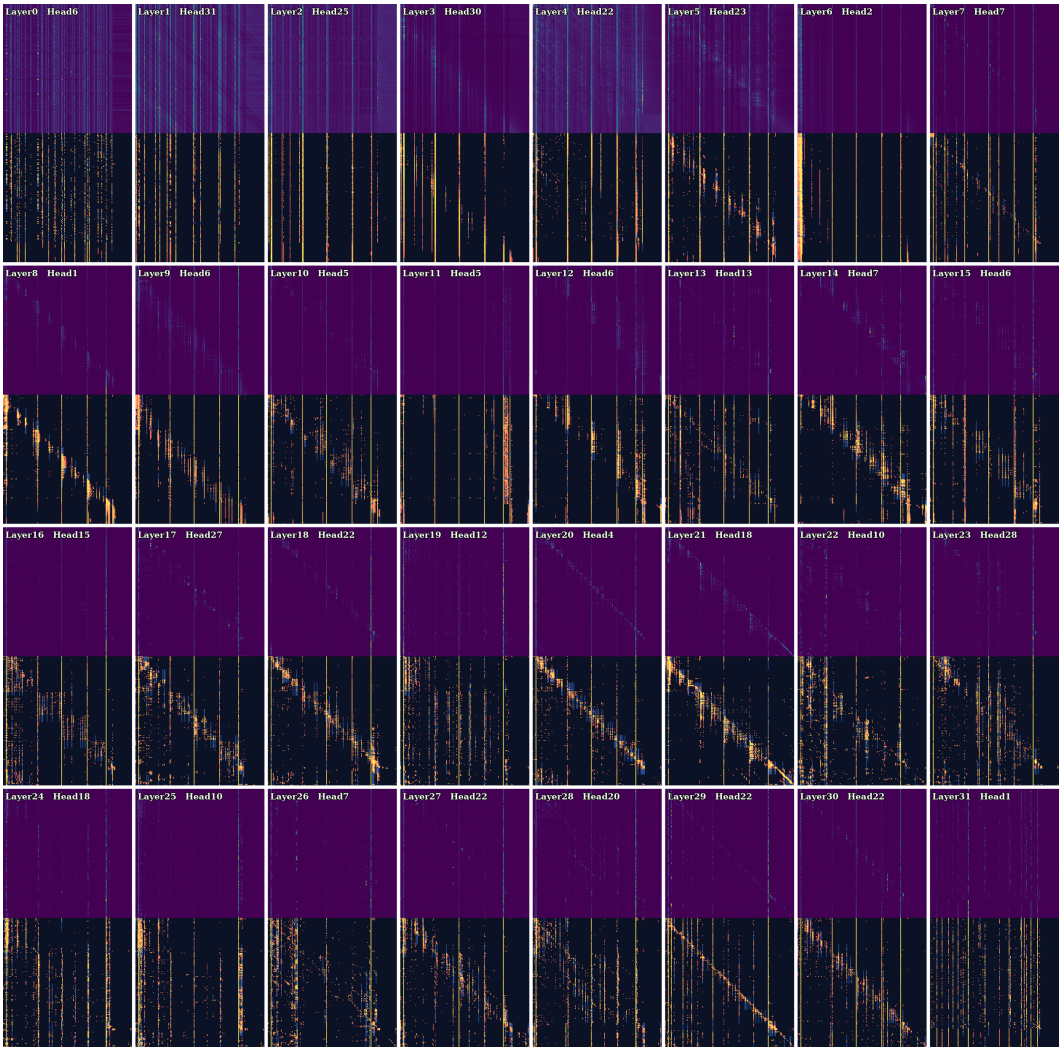


Figure 7: Additional attention visualizations in LLaDA-1.5. Each layer shows one representative head. For each head, we show the attention heatmap and the corresponding per-token top-k positions.

A.2 Quantitative Analysis of Column-Sparse Patterns

We quantitatively analyze column-sparse patterns on HumanEval, GSM8K, and RULER, reporting mean initial-state attention recall at 90% target sparsity. For each query, Oracle Top-k selects the $k = \lceil 0.1n \rceil$ highest-weight keys, where n is the sequence length. Attention recall measures the fraction of Oracle Top-k attention mass retained by the sparse pattern; higher values indicate better preservation of important attention.

Consistency across layers. Figure 8 compares recall across layers in LLaDA-1.5 and Dream. The horizontal axis denotes the layer index, and the vertical axis denotes attention recall; the blue and orange curves correspond to column-sparse and block-sparse patterns, respectively. Column sparsity maintains high recall and consistently outperforms block sparsity at every layer in both models. This indicates that the concentration of important attention on a small subset of columns is prevalent across different model depths.

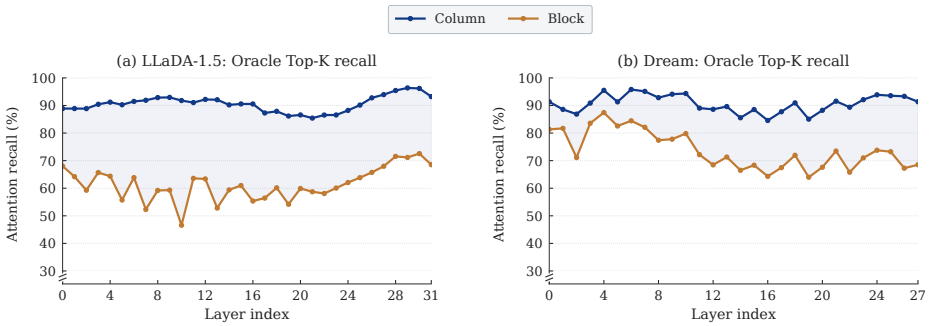


Figure 8: Attention recall across layers. Column sparsity outperforms block sparsity at every layer in LLaDA-1.5 and Dream, demonstrating its ability to preserve important attention throughout model depth.

Consistency across attention heads. Figure 9 further compares recall across attention heads within each layer. The top and bottom rows show LLaDA-1.5 and Dream, respectively, while the left and right columns show column-sparse and block-sparse patterns. Each cell represents an attention head in a particular layer, with greener colors indicating higher recall. The column-sparse heatmaps exhibit broadly high recall across layers and heads, exceeding block-sparse recall for every layer–head pair shown. Thus, the advantage of column sparsity is shared across a broad range of attention heads, rather than being driven by only a few heads.

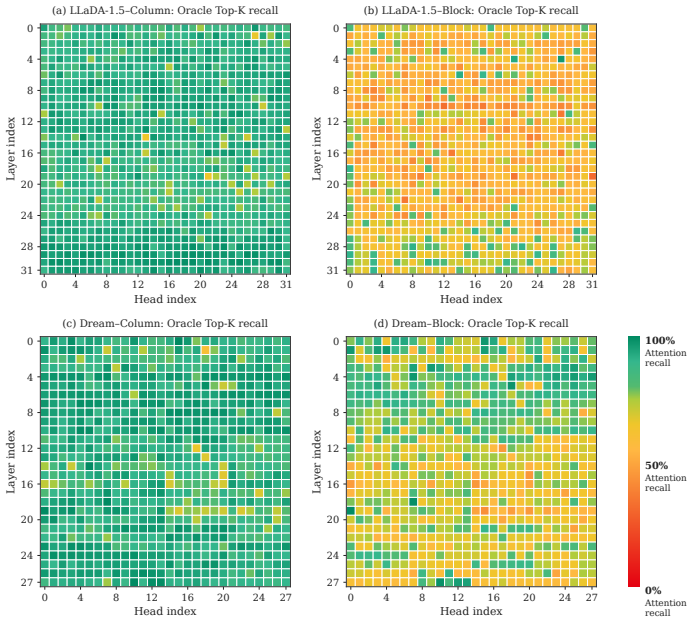


Figure 9: Attention recall across layers and attention heads. Column sparsity achieves broadly high recall across attention heads in both models, highlighting its consistent advantage in preserving important attention.

B COLUMN-SPARSE KERNEL PSEUDOCODE

Algorithm 1 gives the pseudocode of our column-sparse attention kernel. The kernel takes the column indices for each query group as input, and gathers only the corresponding K and V blocks. Attention is then computed on the gathered blocks without materializing the full attention matrix. This pseudocode abstracts away low-level CUDA scheduling details and focuses on the data flow used by PulseCol.

Algorithm 1 Column-Sparse Attention Forward

Require: $Q, K, V \in \mathbb{R}^{N\times d}$ stored in HBM; sparse indices $\mathcal{S} \in \mathbb{N}^{T_q \times n_s}$ stored in HBM; query block size B_M; sparse KV tile size B_N.

$T_q \leftarrow \lceil N/B_M \rceil$
$T_s \leftarrow \lceil n_s/B_N \rceil$
Allocate $O \in \mathbb{R}^{N\times d}$ in HBM.
Divide Q into T_q blocks $Q_1, \ldots, Q_{T_q}$ of size $B_M \times d$.
Divide O into T_q blocks $O_1, \ldots, O_{T_q}$ of size $B_M \times d$.
for $1 \le i \le T_q$ **do**
    Load $Q_i \in \mathbb{R}^{B_M\times d}$ from HBM to SRAM.
    Initialize $m_i \leftarrow (-\infty)^{B_M}$, $\ell_i \leftarrow (0)^{B_M}$, and $\mathrm{acc}_i \leftarrow 0^{B_M\times d}$ in SRAM.
    Load sparse index set $\mathcal{I}_i \leftarrow \mathcal{S}[i] \in \mathbb{N}^{n_s}$ from HBM to SRAM.
    for $1 \le t \le T_s$ **do**
        $I_{i,t} \leftarrow \mathcal{I}_i[(t-1)B_N : tB_N]$
        Gather $K_{I_{i,t}}, V_{I_{i,t}} \in \mathbb{R}^{B_N\times d}$ from HBM to SRAM.
        $R_{i,t} \leftarrow Q_i K_{I_{i,t}}^\top \in \mathbb{R}^{B_M\times B_N}$
        Apply attention scaling to $R_{i,t}$.
        $\widetilde{m}_{i,t} \leftarrow \mathrm{rowmax}(R_{i,t}) \in \mathbb{R}^{B_M}$
        $\widetilde{P}_{i,t} \leftarrow \exp(R_{i,t} - \widetilde{m}_{i,t}) \in \mathbb{R}^{B_M\times B_N}$
        $\widetilde{\ell}_{i,t} \leftarrow \mathrm{rowsum}(\widetilde{P}_{i,t}) \in \mathbb{R}^{B_M}$
        $m_i^{\mathrm{new}} \leftarrow \max(m_i, \widetilde{m}_{i,t})$
        $\ell_i^{\mathrm{new}} \leftarrow \exp(m_i - m_i^{\mathrm{new}}) \odot \ell_i + \exp(\widetilde{m}_{i,t} - m_i^{\mathrm{new}}) \odot \widetilde{\ell}_{i,t}$
        $\mathrm{acc}_i \leftarrow \exp(m_i - m_i^{\mathrm{new}}) \odot \mathrm{acc}_i + \exp(\widetilde{m}_{i,t} - m_i^{\mathrm{new}}) \odot \widetilde{P}_{i,t} V_{I_{i,t}}$
        $\ell_i \leftarrow \ell_i^{\mathrm{new}}, m_i \leftarrow m_i^{\mathrm{new}}$
    end for
    $O_i \leftarrow \mathrm{diag}(\ell_i)^{-1}\mathrm{acc}_i$
    Write $O_i \in \mathbb{R}^{B_M\times d}$ from SRAM to HBM.
end for
return O

C LONG-CONTEXT EVALUATION ON RULER-32K

To evaluate PulseCol on longer contexts, we conduct RULER-32K experiments on UltraLLaDA, which supports longer context lengths. We use the same PulseCol parameter configuration as for LLaDA-1.5. PulseCol also maintains performance close to full attention on these longer contexts.

Table 3: Detailed RULER-32K scores (%) on UltraLLaDA. S, MK, MQ, and MV denote single-needle, multi-key, multi-query, and multi-value retrieval, respectively. VT denotes Variable Tracking; CWE/FWE denote Common/Frequent Words Extraction; QS/QH denote SQuAD/HotpotQA.

	niah								ruler					
	S-1	S-2	S-3	MK-1	MK-2	MK-3	MQ	MV	VT	CWE	FWE	QS	QH	AVG
32K-Length														
UltraLLaDA (Full)	100.00	100.00	98.60	100.00	100.00	100.00	99.80	96.50	100.00	13.58	58.27	61.32	52.20	83.10
+ PulseCol	100.00	100.00	98.40	100.00	100.00	99.40	99.80	98.05	99.88	14.10	56.07	60.68	51.00	82.88

D Additional Efficiency Results

This section provides additional latency results that complement the efficiency analysis in the main paper. We include additional kernel benchmarks, complete LLaDA-1.5 latency results, and long-context latency results on both LLaDA-1.5 and Dream.

D.1 Kernel Speedup with Different Query Group Sizes

Figure 10 reports kernel-level speedups over FlashAttention with additional query group sizes. The column-sparse kernel remains faster across densities and context lengths, with larger gains at higher sparsity and longer context length. Larger query group sizes further improve speedup, as they increase contiguous computation and reduce indexing overhead.

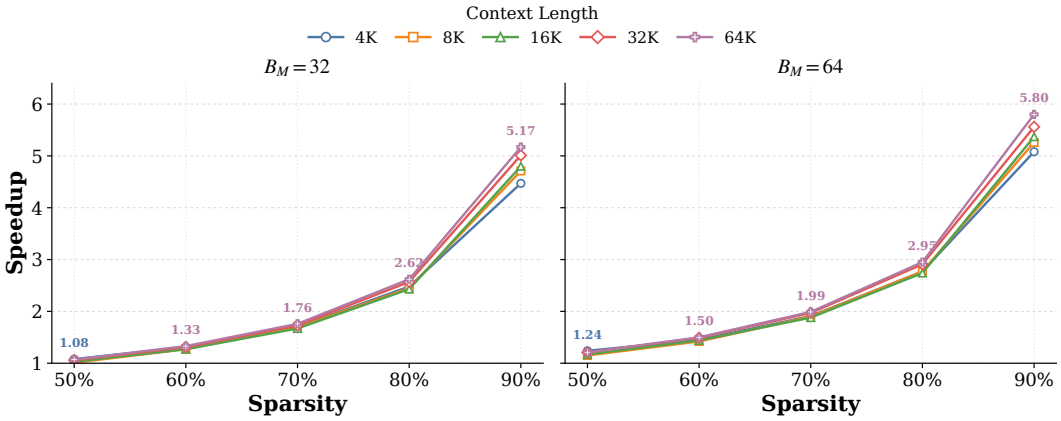


Figure 10: Kernel speedup over FlashAttention with different query group sizes. The two panels report results for query group sizes of 32 and 64 under varying sparsity levels and context lengths.

D.2 End-to-End Latency on LLaDA-1.5

Table 4 reports the complete end-to-end latency results on LLaDA-1.5 across different context lengths and denoising steps. These results complement Figure 1(b), which shows the 1024-step setting in the main paper. PulseCol consistently reduces latency over both the FlashAttention-based full attention baseline and SparseD, with larger gains in long-context settings.

Table 4: Performance comparison across different context lengths and denoising steps. Speedup relative to the baseline is shown in parentheses.

Step	Method	Context Length				
		4k	8k	16k	32k	64k
128	Baseline	31.15	69.25	175.11	471.07	1462.88
	SparseD	30.17 (1.03×)	67.26 (1.03×)	159.92 (1.09×)	396.48 (1.19×)	1143.66 (1.28×)
	Ours	**30.11** (**1.03**×)	**64.59** (**1.07**×)	**148.52** (**1.18**×)	**346.50** (**1.36**×)	**925.63** (**1.58**×)
256	Baseline	64.16	141.77	352.16	945.57	2932.34
	SparseD	61.26 (1.05×)	131.93 (1.07×)	307.00 (1.15×)	749.36 (1.26×)	2101.23 (1.40×)
	Ours	**60.60** (**1.06**×)	**129.15** (**1.10**×)	**285.90** (**1.23**×)	**647.66** (**1.46**×)	**1656.04** (**1.77**×)
512	Baseline	150.45	293.38	712.90	1905.61	5908.74
	SparseD	143.26 (1.05×)	269.22 (1.09×)	609.68 (1.17×)	1470.46 (1.30×)	4046.84 (1.46×)
	Ours	**141.31** (**1.06**×)	**260.43** (**1.13**×)	**564.21** (**1.26**×)	**1256.99** (**1.52**×)	**3138.28** (**1.88**×)
1024	Baseline	324.88	616.40	1477.18	3906.59	11930.43
	SparseD	310.18 (1.05×)	562.63 (1.10×)	1247.98 (1.18×)	2961.99 (1.32×)	7983.49 (1.49×)
	Ours	**304.35** (**1.07**×)	**541.82** (**1.14**×)	**1143.75** (**1.29**×)	**2512.99** (**1.55**×)	**6106.12** (**1.95**×)

D.3 64K Latency on LLaDA-1.5 and Dream

Table 5 and Table 6 report 64K-context latency on LLaDA-1.5 and Dream under different denoising steps and SparseD sparsity settings. PulseCol remains faster across the tested settings, confirming that the long-context efficiency gain holds for both dLLM models.

Table 5: 64K-context latency on LLaDA-1.5 across denoising steps. Latency is reported in seconds, and parentheses denote speedup over FlashAttention.

Method	**64**	**128**	**256**	**512**	**1024**
FlashAttention	726.97	1462.88	2932.34	5908.74	11930.43
SparseD (50%)	741.93 (0.98×)	1304.80 (1.12×)	2432.93 (1.21×)	4708.87 (1.25×)	9321.75 (1.28×)
SparseD (70%)	661.77 (1.10×)	1143.66 (1.28×)	2101.23 (1.40×)	4046.84 (1.46×)	7983.49 (1.49×)
SparseD (80%)	629.64 (1.15×)	1068.86 (1.37×)	1936.38 (1.51×)	3695.13 (1.60×)	7260.83 (1.64×)
Ours (80%)	**559.80 (1.30×)**	**925.63 (1.58×)**	**1656.04 (1.77×)**	**3138.28 (1.88×)**	**6106.12 (1.95×)**

Table 6: 64K-context latency on Dream across denoising steps. Latency is reported in seconds, and parentheses denote speedup over FlashAttention.

Method	**64**	**128**	**256**	**512**	**1024**
FlashAttention	572.05	1144.76	2294.38	4610.00	9348.86
SparseD (50%)	570.00(1.00×)	1010.16(1.13×)	1881.91(1.22×)	3636.74(1.27×)	7206.96(1.30×)
SparseD (70%)	520.36(1.10×)	910.80(1.26×)	1677.17(1.37×)	3217.54(1.43×)	6360.47(1.47×)
SparseD (80%)	484.06(1.18×)	832.12(1.38×)	1517.49(1.51×)	2883.79(1.60×)	5681.51(1.65×)
Ours (80%)	**363.09(1.58 ×)**	**654.55(1.75×)**	**1238.99(1.85×)**	**2417.10(1.91×)**	**4804.59(1.95×)**

E Quality–Efficiency Trade-off

Figure 11 shows the effect of sparsity on accuracy and end-to-end latency. Accuracy on both benchmarks remains stable from 50% to 80% sparsity, while latency decreases by 15.2%. Increasing sparsity to 90% further reduces latency but lowers HumanEval accuracy, suggesting a favorable quality–efficiency trade-off at 80% sparsity.

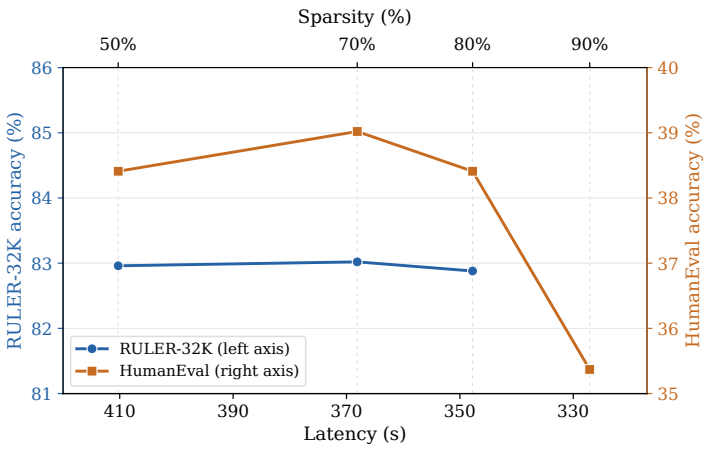


Figure 11: Accuracy–latency trade-off of PulseCol across sparsity levels. Latency is measured on RULER-32K. RULER-32K and HumanEval accuracy use the left and right axes, respectively.

F Additional Ablation on Refresh Schedule

We further study how refresh steps are placed within the early refresh window. All variants use the same temporal coverage ratio η and refresh budget R, and differ only in the refresh schedule. Our default schedule places refreshes uniformly within the early refresh window of $\lfloor \eta T \rfloor$ denoising steps. The random schedule samples the same number of refresh steps uniformly at random from this window. The power-based schedule instead uses normalized positions $(i/(R-1))^2$ for $i = 0, \ldots, R-1$, which concentrates refreshes near the beginning of the window.

As shown in Table 7, the uniform schedule gives a more balanced result across LLaDA-1.5 and Dream. While the power-based schedule slightly improves LLaDA-1.5, it leads to a large drop on Dream. Random refresh is also weaker than the uniform schedule on both models. These results show that the refresh schedule has a clear effect on performance. They also suggest that the uniform schedule is a reasonable default, while more adaptive refresh scheduling may further improve the method.

Table 7: Ablation of refresh schedules on HumanEval.

Method	**ours**	**random**	**power-based**
LLaDA-1.5	38.41	36.58	39.02
Dream	57.32	56.71	52.43

G Limitations

PulseCol currently uses a fixed sparse-reuse schedule, controlled by the refresh window and the number of refreshes. The same configuration is used across all benchmarks, which keeps the method simple and stable. However, our hyperparameter and refresh schedule analyses show that performance can depend on when and how often sparse indices are refreshed. This suggests that a fixed schedule may not be optimal for every model or input. Adaptive refresh strategies based on attention changes during inference may further improve the accuracy-latency trade-off.