HALOBLOCK: FASTER AND BETTER DECODING FOR DIFFUSION LARGE LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion large language models (dLLMs) have attracted increasing research attention due to their bidirectional attention and arbitrary order decoding. However, there are two issues in dLLMs inference: the decoding difficulty is dominated by the tokens on the right side of the block, and the block-wise decoding inherently suffers from boundary contextual discontinuity. To tackle these limitations, we propose HaloBlock, a difficulty-aware adaptive halo mechanism for dLLM inference. We first leverage bidirectional cumulative entropy to characterize block decoding difficulty. Then, based on the bidirectional decoding difficulty, we adaptively allocate bidirectional halo regions on both sides of each block to mitigate boundary contextual discontinuity. The difficulty-aware design of HaloBlock smooths cross-block context propagation and lowers decoding difficulty for hard tokens, yielding better generation quality and faster convergence. Evaluated on LLaDA-8B-Instruct, Dream-7B-Instruct and LLaDA-1.5 across mathematical reasoning and code generation benchmarks, HaloBlock demonstrates consistent accuracy improvements and higher inference efficiency. Source codes will be publicly available.

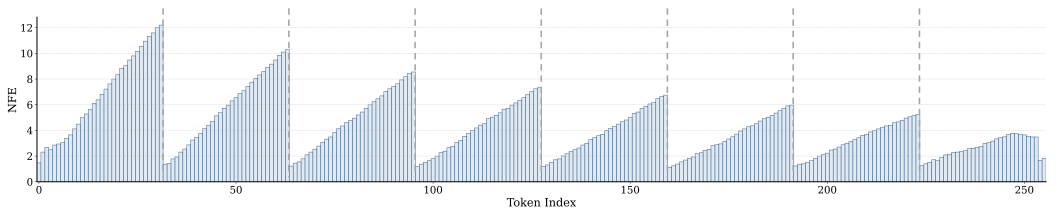
1 INTRODUCTION

Diffusion large language models (dLLMs) (Nie et al., 2025; Ye et al., 2025; Yang et al., 2025) have recently emerged as a promising alternative to the classic autoregressive (AR) LLMs, attracting rapidly growing research attention due to their parallelism. Equipped with inherent bidirectional attention and arbitrary-order decoding, dLLMs exhibit remarkable performance on long-range understanding and controllable editing tasks (Lee et al., 2025; Li et al., 2026b). Despite their architectural advantages, dLLMs suffer from substantially high inference overhead and the convergence difficulty in decoding text. Pioneering works (Arriola et al., 2025; Wu et al., 2026) have established block diffusion for semi-autoregressive decoding and key-value (KV) cache for inference acceleration, both of which have become the standard components in modern dLLM inference.

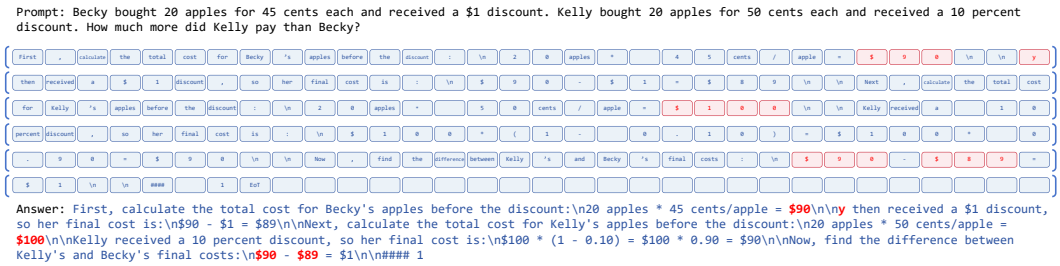
While block-wise decoding successfully alleviates the difficulty of global decoding, it introduces new challenges. The first is the inherent imbalanced difficulty in intra-block decoding. Fig. 1(a) shows the sawtooth-shaped variation in per-token NFE across token index. The left-side tokens receive stronger support from preceding context, which significantly reduces their decoding difficulty, whereas tokens on the right benefit far less from contextual cues and thus face substantially higher decoding difficulty (a failure example is shown in Fig. 1(b)). This position-dependent convergence heterogeneity is broadly consistent with the observations in adaptive parallel decoding literature (Israel et al., 2025; Lu et al., 2026), but its intra-block form in block-wise dLLM inference has not been identified or formalized. Furthermore, as the total forward passes per block are dictated by the hardest tokens, this uneven difficulty distribution becomes a non-trivial issue on inference efficiency. We refer to this imbalanced difficulty in intra-block decoding as left-dominance effect. Second, a less discussed issue is the contextual discontinuity at block boundaries. This discontinuity arises primarily from the asynchronized KV cache update across blocks, a fundamental limitation of block-wise decoding designs (Wu et al., 2026). During intra-block decoding iterations, the KV cache outside the current block is held constant. This forces intra-block to interact with stale context and thus undermines cross-block semantic coherence.

To address the above challenges, we propose HaloBlock, a token-difficulty-aware decoding scheme for dLLMs. Specifically, we first employ bidirectional cumulative entropy as a token-level metric to

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.



(a) Token-wise NFE



(b) Illustrative examples

Figure 1: Illustration of the inference for dLLMs using LLaDA-8B-Instruct. (a) Token-wise Number of Function Evaluations (NFE). (b) An example of mathematical reasoning. The reasoning errors are highlighted in **red bold**. We see the left-dominance effect in block-wise decoding. Zoom in for a better view.

represent the token decoding difficulty. Based on the estimated difficulty, we adaptively allocate halo regions to both ends of each decoding block. The left halo consists of already decoded tokens from preceding blocks, while the right halo is composed of masked placeholder tokens. With bidirectional attention, these halo regions effectively expand the receptive field of the current block and refresh the contextual information at boundaries, alleviating the boundary contextual discontinuity caused by asynchronous KV cache updates. As a result, HaloBlock retains sufficient contextual coverage for high-uncertainty regions, which eases the decoding difficulty and achieves faster decoding convergence. Our contributions are as follows:

- We reveal the key structural limitations, i.e., left-dominance effect and boundary contextual discontinuity, in block-wise decoding of dLLMs.
- We present HaloBlock, a training-free difficulty-aware decoding framework that adaptively allocates halos to each block, which effectively reduces the decoding difficulty and smooths the context generation at block boundaries.
- Experiments on representative dLLM, including LLaDA-8B-Instruct, Dream-7B-Instruct, and LLaDA-1.5, verify the effectiveness of HaloBlock, which yields clear improvements in generation accuracy and even achieves a modest speedup in inference.

2 Related Work

The predominant auto-regressive (AR) language models (Brown et al., 2020; Achiam et al., 2023; Touvron et al., 2023a;b; Bai et al., 2023) have long been used in Natural Language Processing. Recently, diffusion large language models (dLLMs) (Sahoo et al., 2024; Nie et al., 2025; Ye et al., 2025) emerge as a promising alternatives to AR models. Unlike AR models that are constrained to strict left-to-right generation, dLLMs are equipped with bidirectional attention and support arbitrary-order decoding, granting them unique strengths in long-range context understanding and controllable text editing (Lee et al., 2025; Li et al., 2026b). There are mainly two directions to further improve dLLMs: to accelerate the inference speed and to improve the generation quality. Block-diffusion (Arriola et al., 2025) established a semi-AR model by introducing block discrete denoising diffusion. Fast dLLM (Wu et al., 2026) introduced prefix cache and dual cache to enable KV cache reusing for the full bidirectional attention dLLMs.

Beyond the KV cache improvements (Ma et al., 2025; Liu et al., 2026), LLaDA (Nie et al., 2025) firstly demonstrated that a strong reasoning performance can be achieved by training a large-scale dLLM from scratch. For generation quality, it has been found in AdaBlock (Lu et al., 2026) that a

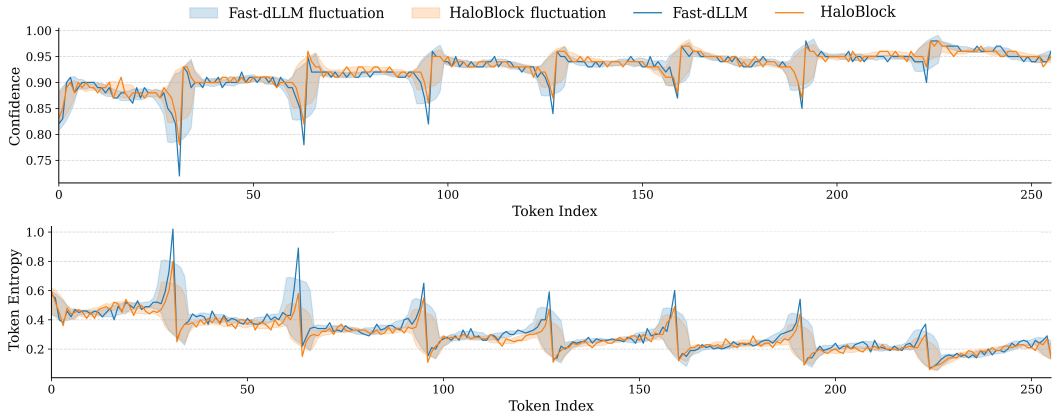


Figure 2: Visualization of token confidence and token entropy on LLaDA-8B-Instruct (GSM8K) with generation length $T_x = 256$ and block size $t = 32$.

reasonable block size is related to the semantic segment of the sequence: better alignment, better reasoning accuracy. Unmasking or re-masking usually requires multiple hyper-parameters and meticulous selection pipeline to converge (Hong et al., 2026; Qi et al., 2026). In addition, researchers have attempted to add some assistant blocks, tokens, or masks to calibrate, supplement, and expand the decoding range (Li et al., 2026a; Chen et al., 2026a).

Apart from the above training-free methods, there are some training approaches to enhance the parallel decoding for dLLMs as well. DParallel (Chen et al., 2026b) performs self-distillation on pre-trained diffusion LLMs to encourage the models to decode more tokens at a time. DMPO (Zhu et al., 2026b) studies a distribution matching policy optimization framework tailored for reinforcement learning fine-tuning of dLLMs. D2F (Wang et al., 2026) introduces discrete diffusion forcing to simultaneously enable accurate block-level KV cache and parallel decoding across multiple blocks through a causally-restricted attention architecture. D3LLM (Qian et al., 2026) employs pseudo-trajectory distillation to guide dLLM inference. Among all the aforementioned methods, a prevailing belief is that block-wise decoding and KV cache have become standard components of dLLMs to improve decoding quality and speed, respectively.

3 Methodology

We start with a standard conditional text generation problem. A sequence X to be generated of length T_x is initialized by a series of [MASK] tokens. Together with the prompt P of length T_p, we form the input sequence $S = [P, X]$. X is decoded iteratively from the [MASK] tokens to the sampled tokens in the vocabulary space $\mathcal{V}$. Block-wise decoding, which was first introduced by (Arriola et al., 2025), partitions the sequence X into a series of sub-sequences X_i, each of which consists of a mount of tokens with length t as the block size. The decoding algorithm is performed on the blocks one by one from left to right. A static KV cache Y is used to interact with X for inference acceleration.

3.1 Empirical Analysis

Block-wise decoding vs. full sequence decoding. Block-wise decoding is semi-autoregressive. It can be seen as an intermediate case between $t = 1$ and $t = T_x$, i.e., $1 \leq t \leq T_x$. Generally speaking, the sequence X can be viewed as a preceding subsequence of a longer generating sequence. Once X is generated, it can be seen as the prefix condition text to generate the subsequent text. From this perspective, each decoding block is functionally equivalent to a miniature full-sequence generation task. Both block-wise decoding and full sequence decoding follow the same discrete diffusion denoising, and they are conditioned on left-side prefix context. This leads to a conclusion: the limitations observed in full-sequence diffusion decoding will not be eliminated by sequence splitting; instead, they will manifest within each individual block on a smaller scale.

For full-sequence dLLM inference, a well-known intrinsic limitation is the position-dependent disparity in per-token decoding difficulty. Because of the sequential nature of natural language,

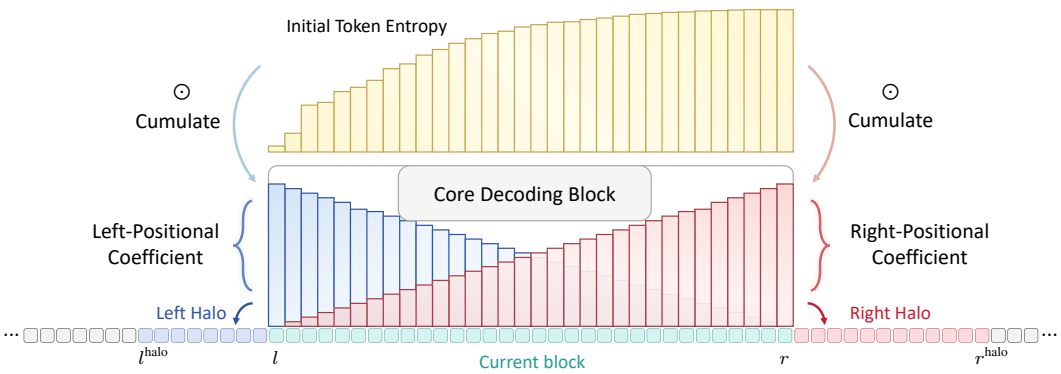


Figure 3: Computational graph of the HaloBlock. The halo lengths are determined by the bidirectional decoding difficulty, which is cumulated by the positional coefficients and the initial token entropy.

tokens near the left side of the sequence benefit from richer accumulated contextual cues from the prefix, which reduces denoising difficulty and leads to faster convergence (Wu et al., 2023). Tokens closer to the end of the sequence, by contrast, have access to far fewer contextual information and thus require more denoising iterations to reach comparable confidence. This uneven difficulty distribution, referred to as the left-dominance effect in this work, is a native property of diffusion-based language generation and becomes more pronounced as the sequence length increases.

Left-dominance effect. In Fig. 1(a), we visualize the per-token Number of Function Evaluations (NFE) when tokens are sampled and unmasked. It shows that NFE grows linearly as token index increases inside a block. Although the confidence-aware parallel decoding strategy (Nie et al., 2025; Wu et al., 2026) favors the high-confidence token more, which does not introduce position bias, the number of forward passes required for token sampling still exhibits a clear left-dominance effect. The tokens near the right-side of the block require much more iteration to converge. This trend holds for all blocks and results in a sawtooth-shaped variation in per-token NFE across token index. As a result, it may easily lead to false prediction near the right edge of the block. An example is illustrated in Fig. 1(b), where the prediction errors occur at a non-trivial frequency in the right area.

Boundary contextual discontinuity. The contextual discontinuity is presumably caused by the asynchronous KV cache updates at the block boundary (Wu et al., 2026). To better analyze this phenomenon, we visualize the token confidence and token entropy for every position of the sequence and average them over all samples. As shown in Fig. 2, there are apparent confidence jumps and entropy jumps at the block boundaries. Notably, token confidence and entropy are relatively stable for most positions in the block, and only jump sharply near the block end. This suggests that the linear growth in NFE merely reflects a gradual increase in per-token decoding difficulty, while accumulated denoising errors manifest abruptly at block boundaries.

3.2 HaloBlock

Motivated by the aforementioned limitations of dLLM decoding, we introduce HaloBlock, a difficulty-aware block-wise decoding framework for dLLM inference. A halo is an auxiliary token region appended to both sides of the decoding block. Halo tokens participate in KV-cache updates during iterative denoising, faciliting dynamic contextual cues update and supporting the current block. Let $X_i = S[l_i : r_i]$ be the i-th block with left edge l_i and right edge r_i. The dLLM $\mathcal{M}$ infers the sequence S and outputs the full logits z and the full KV cache Y. We first introduce bidirectional halo regions to deliver contextual support for the block:

$$l_i^{\text{halo}}, r_i^{\text{halo}} = \texttt{BidirectionalHaloAllocation}(l_i, r_i, z[l_i : r_i]), \quad l_i^{\text{halo}}, r_i^{\text{halo}} \le T_p + T_x, \tag{1}$$

where $z[l_i : r_i]$ represents block logits. In the following, we delve deep into the function of Bidirectional Halo Allocation (Alg. 1).

The core of this allocation function is to quantify decoding difficulty in a bidirectional manner, so that halo lengths on each side can be matched to the actual contextual demand of tokens near the corresponding boundary. We use initial token entropy at the first iterative step as the fundamental measure of per-token decoding difficulty. Given the block logits $z[l_i : r_i]$, we compute the token-level

Algorithm 1 Bidirectional Halo Allocation

Inputs: Block edges l, r, block logits $z \in \mathbb{R}^{t\times|\mathcal{V}|}$. Inherited parameters: sequence $S \in \mathcal{V}^{T_p+T_x}$, temperature τ, halo factor α.

Output: HaloBlock edges $l_{\text{halo}}, r_{\text{halo}}$

$p \leftarrow \text{Softmax}(z)$

$H \leftarrow -\sum p \log p$ ▷ Token entropy

$\text{pos} \leftarrow \{0, 1, \ldots, t-1\}$ ▷ Position indices within the block

$c_{\text{left}} \leftarrow 1 - \text{pos}/(t-1)$; $c_{\text{right}} \leftarrow \text{pos}/(t-1)$ ▷ Linear positional coefficients

$U^{\text{left}} \leftarrow \alpha \sum (H \odot c_{\text{left}})$ ▷ Left halo length

$U^{\text{right}} \leftarrow \alpha \sum (H \odot c_{\text{right}})$ ▷ Right halo length

$l^{\text{halo}} \leftarrow \max\left(0,\ l - \lfloor U^{\text{left}} \rfloor\right)$; $r^{\text{halo}} \leftarrow \min\left(T_p + T_x,\ r + \lfloor U^{\text{right}} \rfloor\right)$

return $l^{\text{halo}}, r^{\text{halo}}$

probability distribution via softmax function, and derive entropy for each position:

$$H[l_i : r_i] = -\sum_{v\in|\mathcal{V}|} p[l_i : r_i](v) \log p[l_i : r_i](v); \quad p[l_i : r_i] = \text{Softmax}(z[l_i : r_i]). \tag{2}$$

To capture the directional demand for context, we introduce a pair of linear positional coefficients for each token, which is given by:

$$c_{\text{left}}[l_i : r_i] = 1 - \text{pos}/(t-1); \quad c_{\text{right}}[l_i : r_i] = \text{pos}/(t-1), \quad \text{pos} = \{0, 1, \ldots, t-1\}. \tag{3}$$

These coefficients assign higher weights to tokens closer to the corresponding block boundary, as tokens near the edge rely more heavily on cross-boundary context for accurate denoising. We then allocate halo length by cumulating the decoding difficulty for both directions on all [MASK] tokens:

$$U_i^{\text{left}} = \alpha \sum (H[l_i : r_i] \odot c_{\text{left}}[l_i : r_i]); \quad U_i^{\text{right}} = \alpha \sum (H[l_i : r_i] \odot c_{\text{right}}[l_i : r_i]), \tag{4}$$

where $\alpha \geq 0$ is introduced to control the scale of halo regions. Finally, the edges of HaloBlock can be represented as:

$$l_i^{\text{halo}} = \max\left(0,\ l - \lfloor U_i^{\text{left}} \rfloor\right); \quad r_i^{\text{halo}} = \min\left(T_p + T_x,\ r + \lfloor U_i^{\text{right}} \rfloor\right), \tag{5}$$

where $\lfloor \cdot \rfloor$ denotes rounding operation. The resulting HaloBlock is represented by $S[l_i^{\text{halo}} : r_i^{\text{halo}}]$ with left edge l_i^{halo} and right edge r_i^{halo}. An overview of HaloBlock computational graph is shown in Fig. 3.

3.3 Block-wise decoding with HaloBlock

Integrated into the block-wise decoding pipeline, HaloBlock alleviates the two issues identified in our empirical analysis (see Sec. 3.1) in a unified manner.

For the intra-block left-dominance effect, the adaptive halo allocation provides asymmetric contextual support: the right side, which suffers from higher decoding difficulty, receives a longer halo and thus more contextual cues to reduce denoising uncertainty. By delivering more context to harder tokens, the difficulty gap within the block is narrowed. Since the total number of forward iterations is determined by the hardest tokens, this leads to faster convergence and lower overall NFE (will be evidenced in the experimental section).

For boundary contextual discontinuity, the extended halo region allows tokens near block edges to attend to a wider range of contextual tokens during iterative denoising. Unlike the standard setting where cross-block KV cache stays static, the halo region updates its KV states along with the current block in each iteration. This synchronized KV cache update alleviates the sharp confidence jump at block boundaries and improves cross-block semantic coherence (refer to the comparison in Fig. 2).

Notably, our HaloBlock is plug-and-play while preserving the original block-wise decoding workflow. The core sampling logic, block scheduling, and KV cache mechanism remain intact, ensuring compatibility with existing dLLM inference optimizations. The full algorithm is provided in **Appendix A**.

Table 1: Quantitative results of dLLM inference on LLaDA-8B-Instruct. The inference time is measured in average over three indepentent runs.

Benchmark	Method	Training-Free	Acc.↑	TPS↑	NFE↓	Time↓
GSM8K (5-shot)	LLaDA		78.5	4.0	256.0	5:19:24
	dParallel (Chen et al., 2026b)		76.4	27.1	32.3	0:40:30
	D2F (Wang et al., 2026)		73.9	56.0	47.4	**0:13:25**
	d3LLM (Qian et al., 2026)		73.5	43.0	**25.8**	0:26:12
	Fast dLLM (Wu et al., 2026)	✓	76.0	58.5	62.8	0:22:24
	AdaBlock (Lu et al., 2026)	✓	78.6	34.5	91.2	0:40:37
	HaloBlock (Ours)	✓	**79.5**	**61.4**	59.5	0:20:46
MATH (4-shot)	LLaDA		33.6	5.6	256.0	15:25:44
	dParallel (Chen et al., 2026b)		31.5	25.0	46.2	2:40:10
	D2F (Wang et al., 2026)		29.4	56.7	84.4	1:26:09
	d3LLM (Qian et al., 2026)		30.7	54.6	**35.4**	**1:16:07**
	Fast dLLM (Wu et al., 2026)	✓	32.5	53.7	92.6	1:36:34
	AdaBlock (Lu et al., 2026)	✓	32.9	34.8	110.4	2:28:41
	HaloBlock (Ours)	✓	**33.2**	**57.2**	88.9	1:30:33
HumanEval (0-shot)	LLaDA		42.1	10.9	256.0	0:15:16
	dParallel (Chen et al., 2026b)		39.6	60.3	**31.6**	**0:01:45**
	D2F (Wang et al., 2026)		41.5	34.5	34.4	0:02:26
	d3LLM (Qian et al., 2026)		39.6	14.3	50.2	0:03:17
	Fast dLLM (Wu et al., 2026)	✓	42.1	55.3	87.0	0:02:53
	AdaBlock (Lu et al., 2026)	✓	39.0	42.1	94.2	0:03:47
	HaloBlock (Ours)	✓	**43.3**	**60.4**	84.5	0:02:38
MBPP (3-shot)	LLaDA		40.6	3.8	256.0	1:38:24
	dParallel (Chen et al., 2026b)		39.8	11.5	**23.2**	0:09:28
	D2F (Wang et al., 2026)		36.6	31.7	31.3	**0:04:14**
	d3LLM (Qian et al., 2026)		36.0	29.3	20.3	0:04:15
	Fast dLLM (Wu et al., 2026)	✓	38.8	37.2	70.6	0:08:33
	AdaBlock (Lu et al., 2026)	✓	40.8	31.2	86.8	0:12:14
	HaloBlock (Ours)	✓	**41.2**	**38.9**	65.7	0:08:15

4 EXPERIMENT

Models and benchmarks. We evaluate HaloBlock on standard dLLMs: LLaDA-8B-Instruct (Nie et al., 2025), Dream-7B-Instruct (Ye et al., 2025), and LLaDA-1.5 (Zhu et al., 2026a). We adopt four widely used benchmarks: GSM8K (5-shot) (Cobbe et al., 2021) and MATH (4-shot) (Hendrycks et al., 2021) for mathematical reasoning, as well as MBPP (3-shot) (Austin et al., 2021) and HumanEval (0-shot) (Chen et al., 2021) for code generation. Generation quality is measured by task accuracy. Generation efficiency is measured by token per second (TPS) and the total inference time for intuitive comparison. We also report average NFE to show the number of model forward pass. All experiments run on four NVIDIA H20 GPUs.

Evaluation details. To provide a fair comparison, we unify the inference pipeline for each model to establish a clean block-wise decoding framework for dLLM inference. For LLaDA series, we use the representative Fast dLLM framework (Wu et al., 2026), and we implement HaloBlock on top of Fast dLLM. For Dream-7B-Instruct, we use the dParallel (Chen et al., 2026b) framework, upon which HaloBlock is implemented. Since different methods may favor different KV cache strategy, we adopt the one that achieves the best task accuracy, which is fixed in comparison. HaloBlock introduces one hyperparameter: the halo factor α in Eqn. (4) to control the scale of halo regions. We conduct a grid search on GSM8K dataset and set $\alpha = 2$, which is fixed in all the evaluation. All other dLLM inference settings remain unchanged.

To conduct a comprehensive comparison, we compare HaloBlock against five state-of-the-art dLLM inference approaches, consisting of three training-based methods: dParallel (Chen et al., 2026b), D2F (Wang et al., 2026), and d3LLM (Qian et al., 2026), as well as two training-free methods: Fast dLLM (Wu et al., 2026) and AdaBlock (Lu et al., 2026). All these competitors represent the most recent advances in efficient dLLM inference. We follow the work of our predecessors and use the optimal hyper-parameters of these methods.

Table 2: Quantitative results of dLLM inference on Dream-7B-Instruct. The inference time is measured in average over three indepentent runs.

Benchmark	Method	Training-Free	Acc.↑	TPS↑	NFE↓	Time↓
GSM8K (5-shot)	Dream		82.9	19.2	256.0	1:13:20
	dParallel (Chen et al., 2026b)		81.3	138.1	41.8	0:10:11
	D2F (Wang et al., 2026)		77.5	50.9	48.7	0:12:06
	d3LLM (Qian et al., 2026)		81.7	162.0	**24.0**	**0:07:58**
	Fast dLLM (Wu et al., 2026)	✓	77.8	88.1	66.1	0:14:46
	AdaBlock (Lu et al., 2026)	✓	77.8	94.1	66.2	0:14:10
	HaloBlock (Ours)	✓	**82.3**	**173.1**	38.0	0:08:08
MATH (4-shot)	Dream		38.3	7.9	256.0	11:16:58
	dParallel (Chen et al., 2026b)		37.3	110.0	66.5	0:48:33
	D2F (Wang et al., 2026)		30.9	52.8	82.8	1:14:55
	d3LLM (Qian et al., 2026)		37.1	37.3	**41.8**	2:23:14
	Fast dLLM (Wu et al., 2026)	✓	37.2	48.4	128.8	1:39:28
	AdaBlock (Lu et al., 2026)	✓	38.4	63.1	103.3	1:24:05
	HaloBlock (Ours)	✓	**38.3**	**122.8**	63.4	**0:43:29**
HumanEval (0-shot)	Dream		55.5	12.6	256.0	0:13:55
	dParallel (Chen et al., 2026b)		53.7	134.5	34.1	0:01:12
	D2F (Wang et al., 2026)		51.2	28.5	107.6	0:03:15
	d3LLM (Qian et al., 2026)		56.1	68.4	35.5	0:02:25
	Fast dLLM (Wu et al., 2026)	✓	51.2	37.2	89.6	0:02:06
	AdaBlock (Lu et al., 2026)	✓	54.9	61.5	68.7	0:02:44
	HaloBlock (Ours)	✓	**56.1**	**149.0**	**31.6**	**0:01:06**
MBPP (3-shot)	Dream		57.6	17.1	256.0	0:30:34
	dParallel (Chen et al., 2026b)		53.4	187.0	27.1	0:02:50
	D2F (Wang et al., 2026)		53.2	36.3	67.7	0:05:36
	d3LLM (Qian et al., 2026)		54.8	111.9	31.8	0:04:46
	Fast dLLM (Wu et al., 2026)	✓	52.8	30.7	53.3	0:05:09
	AdaBlock (Lu et al., 2026)	✓	**55.0**	101.0	51.0	0:04:59
	HaloBlock (Ours)	✓	54.6	**214.5**	**27.0**	**0:02:24**

4.1 Experimental Results

Main results. We conduct a series of experiments on different dLLMs and various decoding settings. The results of LLaDA-8B-Instruct are summarized in Tab. 1. We see that HaloBlock balances speed and accuracy very well across all the four benchmarks, outperforming both training-based and training-free efficient dLLM inference methods. Specifically, HaloBlock produces the best task accuracy among all the advanced strategies of dLLM inference. Comparing to the base model Fast dLLM, our method consistently improves the task accuracy by +3.5%, +0.7%, +1.2%, and +2.4% on GSM8K, MATH, HumanEval, and MBPP benchmarks, respectively. Notably, HaloBlock even achieves faster inference speed and decoding convergence speed. For example, compared to Fast dLLM, TPS is improved by 2.9 token/s on GSM8K with 3.3 lower average NFE. The inference time is notably reduced by 7.3% from 0:22:24 to 0:20:46. Similarly, Tab. 2 reports quantitative results on Dream-7B-Instruct. In terms of accuracy, HaloBlock remains highly competitive with the vanilla Dream model. Compared to the baseline dParallel, HaloBlock produces +1.0, +1.0, +2.4, +1.2 accuracy gains on GSM8K, MATH, HumanEval, and MBPP, respectively. For inference efficiency, compared to existing training-free methods, HaloBlock consistently delivers the highest TPS and fastest inference time across all four benchmarks. Those training-based approaches that achieve acceleration by reducing the number of model forward passes; however, they generally suffer from non-negligible accuracy degradation. By contrast, HaloBlock yields a more favorable speed-accuracy trade-off for dLLM inference. For the results of LLaDA-1.5, kindly refer to **Appendix B**.

HaloBlock eases the boundary contextual discontinuity. To verify how HaloBlock takes effect on block-wise decoding, we visualize per-token denoising confidence and token entropy over a 256-token generation sequence. As shown in Fig. 2, we observe consistent and abrupt discontinuities at every block boundary: denoising confidence drops sharply and token entropy spikes upward, as tokens adjacent to the partition lose direct connection to the new contextual information from the neighboring block. These jumps mark positions where decoding uncertainty rises abruptly, which corresponds to where most generation errors occur in block-wise dLLM. HaloBlock effectively

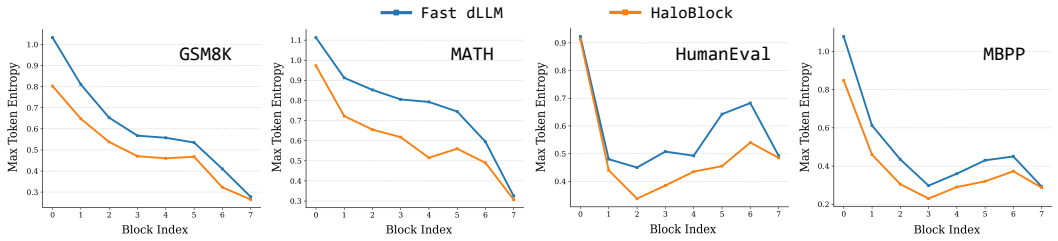


Figure 4: Visualization of the max token entropy for each block when the tokens are unmasked. HaloBlock reduces the max token entropy, which indicates lower uncertainty.

Table 3: Quantitative results of dLLM inference with various generation length T_x and block size t. The inference time is measured in average over three indepentent runs.

Method	T_x	$t = 8$			$t = 16$			$t = 32$		
		Acc.↑	**TPS↑**	**Time↓**	**Acc.↑**	**TPS↑**	**Time↓**	**Acc.↑**	**TPS↑**	**Time↓**
LLaDA	256	80.5	4.4	4:49:38	79.2	4.5	4:48:55	78.5	4.0	5:19:24
Fast dLLM	256	78.9	24.1	0:52:40	77.4	35.9	0:35:32	76.0	58.5	0:22:24
HaloBlock	256	80.7	25.5	0:50:11	79.8	38.7	0:33:00	79.5	61.4	0:20:46
LLaDA	512	80.4	2.1	11:30:26	77.7	2.1	11:31:15	76.1	2.5	10:14:10
Fast dLLM	512	76.7	15.5	1:36:27	76.9	25.5	0:56:01	74.5	40.4	0:37:17
HaloBlock	512	78.8	15.6	1:35:38	79.5	27.1	0:54:35	77.5	42.7	0:34:53

smooths these boundary-induced fluctuations. With adaptive halo regions extending the attention window across partitions, the magnitude of confidence drops and entropy spikes at boundaries is substantially reduced, yielding a more stable and continuous generation across the entire sequence.

HaloBlock accelerates decoding convergence by reducing decoding difficulty. One might expect that attaching the effective context window via halo regions would incur higher computational overhead per forward pass. To further understand the source of HaloBlock's efficiency gains, we visualize the maximum token entropy within each block across 8 consecutive blocks, evaluated on four benchmarks: GSM8K, MATH, HumanEval, and MBPP. As shown in Fig. 4, across all four datasets, HaloBlock consistently yields lower maximum token entropy for every block compared to the Fast dLLM baseline. In block-wise decoding, the convergence speed of each block is highly correlated with the token of the highest uncertainty, as this token acts as the rate-limiting step for the entire block. The reduction in maximum per-block token entropy indicates that HaloBlock effectively lowers the decoding difficulty of the hardest tokens within each block, which directly leads to faster convergence. As a result, we observe that HaloBlock reaches acceptable denoising quality with fewer model forward passes compared to the baseline, which explains why HaloBlock achieves higher throughput despite operating with a slightly larger effective attention scope: it trades a modest increase in per-step compute for a substantial reduction in the number of iterative steps. More analyses of HaloBlock can be found in **Appendix C**.

4.2 Ablation Study

Performance across different generation length. We experiment HaloBlock under two widely adopted generation lengths, 256 and 512, to verify its robustness across output length. As shown in Tab. 3, doubling the sequence length reduces throughput for all three methods, as each forward pass operates on a longer sequence and incurs higher computational cost. Across generation length 256 and 512, HaloBlock boosts the generation quality while achieving inference speedup. Meanwhile, the throughput advantage of HaloBlock over Fast dLLM remains consistent across lengths, indicating that the overhead of halo allocation is negligible and does not scale with total sequence length.

Performance across different block size. We further examine how decoding performance varies with block size $t \in \{8, 16, 32\}$. It can be seen from Tab. 3 that larger block sizes yield higher throughput, since more tokens are denoised in parallel per forward evaluation. However, larger blocks also cause accuracy degradation: when t increases from 8 to 32, the task accuracy of Fast dLLM drops 2.9 at $T_x = 256$ and 2.2 at $T_x = 512$. HaloBlock effectively improves this accuracy-speed balance. Across all tested block sizes, it recovers the majority of the accuracy drops, while retaining or even improving inference efficiency.

Table 4: Ablation study on halo factor α. The results are reported by task accuracy and inference time.

Model	Baseline	$\alpha = 1$	$\alpha = 2$	$\alpha = 3$	$\alpha = 4$
LLaDA	76.0	78.0	79.5	79.2	78.2
	0:22:24	0:20:32	0:20:46	0:21:22	0:22:17
Dream	81.3	81.8	82.3	82.0	81.8
	0:10:11	0:07:49	0:08:08	0:08:44	0:08:48

Table 5: Performance of HaloBlock upon more block-wise decoding methods.

Model	Acc.↑	TPS↑	NFE↓	Time↓
AdaBlock	78.6	34.5	91.2	0:40:37
AdaBlock + Ours	79.4	35.6	87.5	0:39:33
dParallel	81.3	138.1	41.8	0:10:11
dParallel + Ours	82.3	173.1	38.0	0:08:08

Effect of halo factor α. In Eqn. (4), HaloBlock introduces one hyper-parameter α to control the scale of halo regions. We investigate the effect of HaloBlock under various halo factors. Tab. 6 reports the experimental results on LLaDA-8B-Instruct and Dream-7B-Instruct. We can see that all the choices of α improve the performance of dLLM. In practice, we empirically set $\alpha = 2$ for all models.

Halo region: difficulty-aware vs. static. We perform a grid search over static halo configurations, where left and right halo lengths are varied respectively from 1 to 16 tokens. As shown in Fig. 5, static halos improve over the 76.0 baseline across most settings, with peak accuracy at around $(9, 10)$. Our HaloBlock reaches 79.5% accuracy without the need of laboriously tuning the left and the right halo length. An intriguing finding is the strong asymmetry between left and right halo sensitivity. A short right halo ($U^{\text{right}} \leq 5$) leads to marked accuracy drops, and even worse performance than having no halo at all in some configurations. This asymmetry presumably stems from the directional nature of sequence generation. The left boundary of a block adjoins already decoded tokens with fully available context, thus it is less dependent on halo extension. The right boundary faces mask tokens, where native contextual information is inherently scarce, making adequate right halo coverage critical for stable decoding. Our difficulty-aware approach adaptively prioritizes high-difficulty positions and allocates halo capacity efficiently.

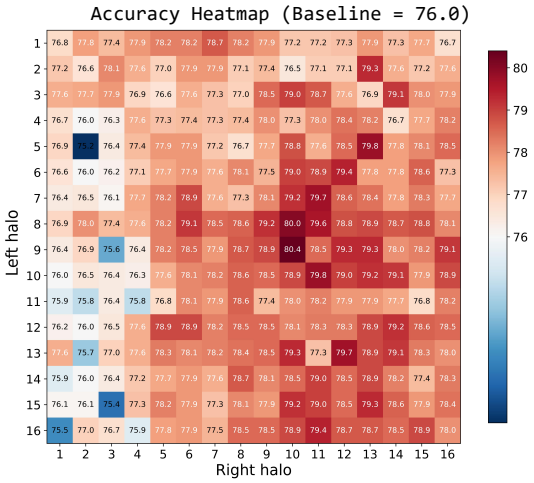


Figure 5: Visualization of accuracy heatmap with static halo region on LLaDA-8B-Instruct (GSM8K).

Compatibility of HaloBlock. We then verify that HaloBlock is orthogonal to some recent dLLM inference optimizations. For example, adaptive block size methods such as AdaBlock (Lu et al., 2026), dynamically assigns block sizes via semantic delimiters. Due to its better alignment between block size and semantic step, AdaBlock can effectively enhance the sampling quality. We then adapt HaloBlock to AdaBlock. The results in Tab. 5 show that further improvements can be achieved in both generation quality and inference efficiency. These results show the consistent effectiveness of HaloBlock: better generation accuracy, faster decoding convergence, and lower inference cost.

5 Conclusion

In this paper, we presented HaloBlock, a simple yet effective method to improve the inference speed and accuracy of dLLMs by alleviating the issues of left-dominance effect and boundary contextual discontinuity. HaloBlock quantifies token decoding difficulty via bidirectional cumulative entropy and use it to adaptively appending halo regions to both sides of each block. Extensive evaluations on LLaDA and Dream models demonstrated that HaloBlock achieves consistent generation quality improvement, as well as faster decoding convergence and lower inference time cost. This work has implications for how we model token decoding difficulty and design contextual support algorithms in block-wise diffusion large language model inference.

Limitations. While simple and effective, as many other dLLM acceleration methods, HaloBlock still relies on block-wise decoding and KV cache. In addition, currently we apply HaloBlock to the popular LLaDA, LLaDA-1.5, and Dream models. In the future, we will adapt HaloBlock to the recently developed dLLMs such as LLaDA-2.0 (Bie et al., 2025), SDAR (Cheng et al., 2026), d3LLM (Qian et al., 2026) and DMPO (Zhu et al., 2026b), etc.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Marianne Arriola, Aaron Gokaslan, Justin Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. In *ICLR*, pp. 50726–50753, 2025.

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

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

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.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *NeurIPS*, pp. 1877–1901, 2020.

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.

Xinhua Chen, Sitao Huang, Cong Guo, Chiyue Wei, Yintao He, Jianyi Zhang, Hai Li, and Yiran Chen. DPad: Efficient diffusion language models with suffix dropout. In *ICLR*, pp. 57286–57316, 2026a.

Zigeng Chen, Gongfan Fang, Xinyin Ma, Ruonan Yu, and Xinchao Wang. dParallel: Learnable parallel decoding for dLLMs. In *ICLR*, pp. 53193–53211, 2026b.

Shuang Cheng, Yihan Bian, Dawei Liu, Yuhua Jiang, Yihao Liu, Linfeng Zhang, Qian Yao, Zhongbo Tian, Wenhai Wang, Qipeng Guo, et al. SDAR: A synergistic diffusion-autoregression paradigm for scalable sequence generation. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 22058–22075, 2026.

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.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. In *NeurIPS*, 2021.

Feng Hong, Geng Yu, Yushi Ye, Haicheng Huang, Huangjie Zheng, Ya Zhang, Yanfeng Wang, and Jiangchao Yao. Wide-in, narrow-out: Revokable decoding for efficient and effective dllms. In *ICLR*, pp. 40132–40150, 2026.

Daniel Israel, Guy Van den Broeck, and Aditya Grover. Accelerating diffusion llms via adaptive parallel decoding. *NeurIPS*, pp. 52870–52888, 2025.

Che Hyun Lee, Heeseung Kim, Jiheum Yeom, and Sungroh Yoon. EdiText: Controllable coarse-to-fine text editing with diffusion language models. In *ACL*, pp. 22798–22815, 2025.

Jinsong Li, Xiaoyi Dong, Yuhang Zang, Yuhang Cao, Jiaqi Wang, and Dahua Lin. Beyond fixed: Training-free variable-length denoising for diffusion large language models. In *ICLR*, pp. 91715–91731, 2026a.

Shufan Li, Yuchen Zhu, Jiuxiang Gu, Kangning Liu, Zhe Lin, Yongxin Chen, Molei Tao, Aditya Grover, and Jason Kuen. LaViDa-R1: Advancing reasoning for unified multimodal diffusion language models. In *ICML*, 2026b.

Zhiyuan Liu, Yicun Yang, Yaojie Zhang, Junjie Chen, Chang Zou, Qingyan Wei, Shaobo Wang, Yichen Zhu, and Linfeng Zhang. dLLM-Cache: Accelerating diffusion large language models with adaptive caching. In *ICML*, 2026.

Guanxi Lu, Hao Chen, Yuto Karashima, Zhican Wang, Daichi Fujiki, and Hongxiang Fan. AdaBlock-dLLM: Semantic-aware diffusion llm inference via adaptive block size. In *ICLR*, volume 2026, pp. 140018–140033, 2026.

Xinyin Ma, Runpeng Yu, Gongfan Fang, and Xinchao Wang. dKV-Cache: The cache for diffusion language models. In *NeurIPS*, pp. 149009–149033, 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. In *NeurIPS*, pp. 50608–50646, 2025.

Xiaojing Qi, Lun Du, Xinyuan Zhang, Lanning Wei, Tao Jin, and Da Zheng. Hierarchy decoding: A training-free parallel decoding strategy for diffusion large language models. In *ICLR*, 2026.

Yu-Yang Qian, Junda Su, Lanxiang Hu, Peiyuan Zhang, Zhijie Deng, Peng Zhao, and Hao Zhang. d3LLM: Ultra-fast diffusion LLM using pseudo-trajectory distillation. In *ICML*, 2026.

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.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

Xu Wang, Chenkai Xu, Yijie Jin, Jiachun Jin, Hao Zhang, Kai Yu, and Zhijie Deng. Diffusion llms can do faster-than-ar inference via discrete diffusion forcing. In *ICLR*, pp. 2282–2299, 2026.

Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. Fast-dLLM: Training-free acceleration of diffusion llm by enabling kv cache and parallel decoding. In *ICLR*, pp. 57027–57051, 2026.

Tong Wu, Zhihao Fan, Xiao Liu, Hai-Tao Zheng, Yeyun Gong, Jian Jiao, Juntao Li, Jian Guo, Nan Duan, Weizhu Chen, et al. AR-Diffusion: Auto-regressive diffusion model for text generation. *NeurIPS*, pp. 39957–39974, 2023.

Ling Yang, Ye Tian, Bowen Li, Xinchen Zhang, Ke Shen, Yunhai Tong, and Mengdi Wang. MMaDA: Multimodal large diffusion language models. In *NeurIPS*, pp. 138867–138907, 2025.

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:2509.01142*, 2025.

Fengqi Zhu, Rongzhen Wang, Shen Nie, Xiaolu Zhang, Chunwei Wu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Llada 1.5: Variance-reduced preference optimization for large language diffusion models. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11425–11460, 2026a.

Yuchen Zhu, Wei Guo, Jaemoo Choi, Petr Molodyk, Bo Yuan, Molei Tao, and Yongxin Chen. Enhancing reasoning for diffusion LLMs via distribution matching policy optimization. In *ICML*, 2026b.

Algorithm 2 HaloBlock

Inputs: Model $\mathcal{M}$, prompt sequence $P \in \mathcal{V}^{T_p}$, refinement steps s, generation length T_x, block size t, the number of blocks b, temperature τ, halo factor α.
Output: Generated full sequence $S = [P, X] \in \mathcal{V}^{T_p+T_x}$

$S \leftarrow [P, \underbrace{[\texttt{MASK}], \cdots, [\texttt{MASK}]}_{T_x}]$ ▷ Initialize sequence
for $i = 1$ **to** b **do**
  $l \leftarrow T_p + (i-1) \cdot t; \quad r \leftarrow l + t$ ▷ Left and right edges of block X_i
  $\mathcal{I} \leftarrow (S[l:r] = [\texttt{MASK}])$
  $\{n_k\}_{k=1}^{s} \leftarrow \texttt{AllocQuota}(\mathcal{I}, s)$ ▷ Allocate unmasking quota per step
  $\mathcal{R} \leftarrow \mathbf{1}^{T_p+T_x}$ ▷ KV cache update indicator
  $z, Y \leftarrow \mathcal{M}(S, \mathcal{R})$ ▷ Obtain full logits and full KV cache
  $S \leftarrow \texttt{Sampling}(z[l:r], \mathcal{I}, S, n_1)$ ▷ Decode and update X
  $l^{\text{halo}}, r^{\text{halo}} = \texttt{BidirectionalHaloAllocation}(l, r, z[l:r])$
  $\mathcal{R} \leftarrow \mathbf{0}^{T_p+T_x}; \quad \mathcal{R}[l^{\text{halo}} : r^{\text{halo}}] \leftarrow \mathbf{1}$ ▷ Identify KV cache update indicator
  for $j = 2$ **to** s **do**
    $\mathcal{I} \leftarrow (S[l:r] = [\texttt{MASK}])$
    if $\sum \mathcal{I} = 0$ **then**
      break ▷ Early exit when block is fully unmasked
    end if
    $z_j, Y \leftarrow \mathcal{M}(S[l^{\text{halo}} : r^{\text{halo}}], Y, \mathcal{R})$ ▷ Block-level logits with updated KV cache
    $S \leftarrow \texttt{Sampling}(z_j[l - l^{\text{halo}} : r - l^{\text{halo}}], \mathcal{I}, S, n_j)$ ▷ Decode and update X
  end for
end for
return S

Appendix

A Block-wise decoding with HaloBlock

We present the formal algorithm for the proposed HaloBlock in Alg. 2, where we adopt Fast dLLM (Wu et al., 2026) as an example baseline framework.

B Experimental results on LLaDA-1.5

We compare HaloBlock with the training-free dLLM inference methods, Fast dLLM (Wu et al., 2026) and AdaBlock (Lu et al., 2026) on LLaDA-1.5. The quantitative results are presented in Tab. 6. Compared to Fast dLLM and AdaBlock, HaloBlock yields accuracy gains and better efficiency under all the four benchmarks. This again demonstrates the effectiveness of our proposed method.

C More Analysis on HaloBlock

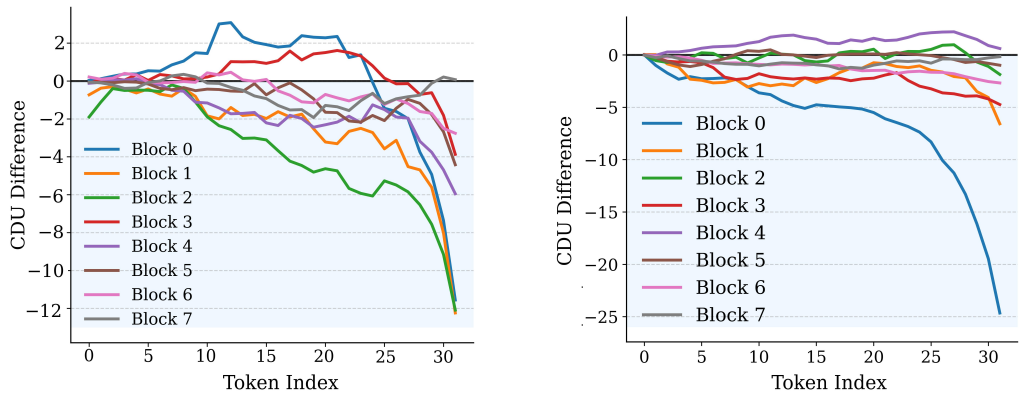
HaloBlock eases the decoding difficulty. To further analyze the behavior of using HaloBlock on dLLM inference, we introduce cumulative decoding uncertainty (CDU) to represent the decoding difficulty for the sequence $S[l:r]$,

$$\text{CDU}[l:r] = \sum_{j=1}^{s} ||H_j[l:r] \cdot \mathcal{I}_j||_2, \quad j \in \{1, 2, \cdots, s\}, \tag{6}$$

where $\mathcal{I}_j = (S_j[l:r] = [\texttt{MASK}])$ denotes the iterative indicator of the sequence $S[l:r]$ checking whether the tokens are unmasked. Therefore, CDU calculates the cumulative token entropy from the

Table 6: Quantitative results of dLLM inference on LLaDA-1.5. The inference time is measured in average over three indepentent runs.

Benchmark	Method	Training-Free	Acc.↑	TPS↑	NFE↓	Time↓
GSM8K (5-shot)	LLaDA-1.5 (Zhu et al., 2026a)		80.5	3.8	256.0	5:14:26
	Fast dLLM (Wu et al., 2026)	✓	80.3	46.3	84.0	0:25:15
	AdaBlock (Lu et al., 2026)	✓	79.8	31.7	88.1	0:37:06
	HaloBlock (Ours)	✓	**81.1**	**49.4**	**79.4**	**0:24:03**
MATH (4-shot)	LLaDA-1.5 (Zhu et al., 2026a)		33.2	5.2	256.0	13:12:47
	Fast dLLM (Wu et al., 2026)	✓	32.5	40.6	106.5	1:51:09
	AdaBlock (Lu et al., 2026)	✓	33.0	42.9	108.1	2:04:13
	HaloBlock (Ours)	✓	**33.3**	**45.9**	**100.3**	**1:38:10**
HumanEval (0-shot)	LLaDA-1.5 (Zhu et al., 2026a)		43.3	4.4	256.0	0:12:23
	Fast dLLM (Wu et al., 2026)	✓	35.4	15.1	106.2	0:03:57
	AdaBlock (Lu et al., 2026)	✓	37.8	12.3	123.5	0:04:05
	HaloBlock (Ours)	✓	**39.0**	**18.0**	**101.3**	**0:03:13**
MBPP (3-shot)	LLaDA-1.5 (Zhu et al., 2026a)		39.2	1.4	256.0	1:27:34
	Fast dLLM (Wu et al., 2026)	✓	35.2	20.2	45.3	0:06:06
	AdaBlock (Lu et al., 2026)	✓	**37.6**	14.1	52.0	0:09:04
	HaloBlock (Ours)	✓	**37.6**	**25.4**	**42.7**	**0:05:25**



(a) LLaDA (b) Dream

Figure 6: Visualization of CDU difference for every block. The difference is calculated by subtracting the CDU of our HaloBlock from the CDU of Fast dLLM. The block size $t = 32$.

beginning to the final unmasking. For easy tokens, the unmasking is quickly achieved, resulting in a low CDU, while for hard tokens, CDU is large as they need multiple iteration to converge.

To showcase how our method affects decoding difficulty, we visualize the CDU difference between our HaloBlock and Fast dLLM. We calculate the CDU difference by subtracting the CDU of our HaloBlock from the CDU of Fast dLLM. As shown in Fig. 6, when the token approaches the right edge of the block, our method can generally produce lower CDU as the CDU difference decreases to negatives. This indicates that our method can effectively reduce the decoding difficulty of hard tokens located on the right side of the block. As a result, faster decoding convergence is achieved. The average NFE is reduced as Tab. 1, Tab. 2, and Tab. 6 show.

HaloBlock is hyper-parameter free. In Fig. 5 of the main paper, we plot the accuracy heatmap with static halo region on GSM8K dataset. We here supplement the accuracy heatmap on MBPP (Austin et al., 2021). Comparing the two accuracy heatmaps in Fig. 7(a) and (b), one can see that the optimal halo configuration on GSM8K may not be the optimum on MBPP. Particularly, the accuracy peak of GSM8K is at $(9, 10)$, which intriguingly turns out to be a performance degradation point on MBPP, yielding an accuracy of 37.6% even lower than the baseline 38.8%. This observation demonstrates that static fixed-size halo configurations cannot generalize across different tasks, as optimal halo lengths are task-dependent and cannot be manually pre-tuned for all downstream scenarios. By

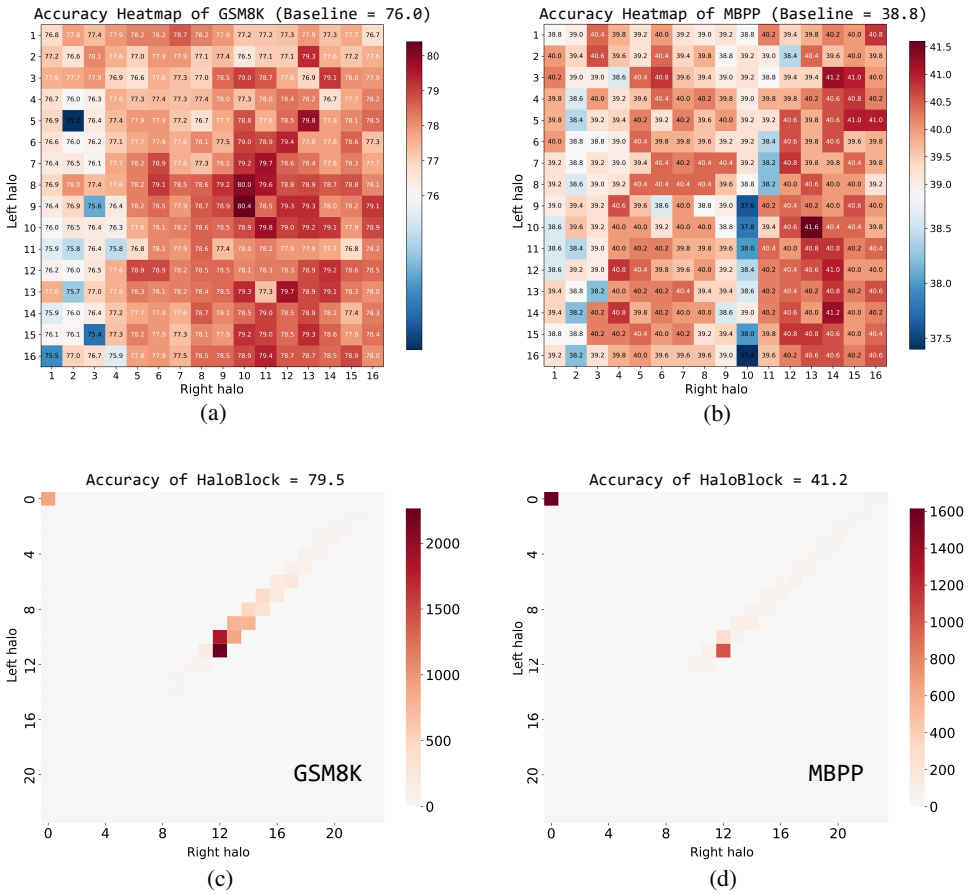


Figure 7: Visualization of accuracy heatmap with static halo region and the statistical count of halo length by our HaloBlock. Our adaptive halo allocation generally performs better than static halos.

comparison, our adaptive halo allocation dynamically determines left- and right-halo lengths based on bidirectional token decoding difficulty. It obviates dataset-specific hyperparameter tuning and achieves robust performance across different reasoning and code-generation benchmarks.

Statistical count of halo length. We plot the statistical count of halo length on GSM8K and MBPP in Fig. 7(c) and (d). It shows that our HaloBlock produces dynamic halo length. For easy tokens, the halo lengths are mostly fall into 0, because the cumulative entropies approach 0. For difficult tokens, the model's output probability distribution is more dispersed, leading to higher entropy. HaloBlock therefore allocates larger halo length for these hard tokens, which account for a non-trivial proportion of all tokens. In addition, an asymmetric distribution of halo allocation can also be observed: the right halo length is generally longer than the left counterpart, due to the left-dominance effect.

Visualization. We showcase three mathematical reasoning cases of dLLM inference in Fig. 8, Fig. 9, and Fig. 10. We also visualize the dynamic halo length for every block. The changes in the halo region once again exhibit asymmetry, with the right halo being generally longer. Then, we compare the inference results produced by our HaloBlock and the baseline Fast dLLM (Wu et al., 2026). The examples show that HaloBlock benefits from the halo region, which allow the current block to see more contextual information and reduce reasoning errors.

AI USE STATEMENT

In this work, we used generative AI tools for polish writing and searching related works. Additionally, we used generative AI tools for analyzing the codes. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work has no ethic issue.

REPRODUCIBILITY STATEMENT

The source codes of this work will be publicly available.

Prompt: Jennie is helping at her mom's office. She has a pile of 60 letters needing stamps, and a pile of letters already stamped. She puts stamps on one-third of the letters needing stamps. If there are now 30 letters in the pile of already-stamped letters, how many were in that pile when Jennie began?



Note: In the above figure, the reasoning errors are highlighted in red and the missing predictions are highlight in **bold black**.

Answer by Fast dLLM: To determine how many letters were in the pile of already-stamped letters when Jennie began, we need to follow these steps:

1. CalculateCalculate the number of letters Jennie stamped.
2. Subtract the number of letters she stamped from the total number of letters needing stamps to find out how many letters are left unstamped.

First, calculate the number of letters Jennie stamped:

$$\text{Number of letters stamped} = \frac{1}{3} \times 60 = 20$$

Next, subtract the number of letters she stamped from the total number of letters needing stamps to find out how many letters are left unstamped:

$$\text{Number of letters left unstamped} = 60 - 20 = 40$$

We are given that there are now 30 letters in the pile of already-stamped letters. Therefore, the number of letters in the pile when Jennie began must have been:

$$\text{Number of letters in the pile when} = 30 - 20 = 0$$

So, there were no letters in the pile of already-stamped letters when Jennie began.

Answer by HaloBlock: To determine how many letters were in the pile of already-stamped letters when Jennie began, we need to follow these steps:

1. Calculate the number of letters Jennie stamps.
2. Determine how many letters were already stamped when Jennie began.

First, let's find out how many letters Jennie stamps. She puts stamps on one-third of the 60 letters needing stamps:

$$\frac{1}{3} \times 60 = 20$$

So, Jennie stamps 20 letters.

Next, we know that after stamping these letters, there are now 30 letters in the pile of already-stamped letters. Let x be the number of letters in the pile of already-stamped letters when Jennie began. After stamping 20 letters, the total number of letters in the pile is:

$$x + 20 = 30$$

Solving for x:

$$x = 30 - 20 = 10$$

Therefore, there were 10 letters in the pile of already already letters when Jennie began.

10

Figure 8: **Example 1**: Comparison of inference results between the baseline Fast dLLM and HaloBlock. We visualize the dynamic halo length in colors. Reasoning errors are highlighted in red.

Prompt: Janeth borrowed $2000 and promised to return it with an additional 10% of the amount. If she is going to pay $165 a month for 12 months, how much will be Janeth's remaining balance by then?



Note: In the above figure, the reasoning errors are highlighted in red and the missing predictions are highlight in **bold black**.

Answer by Fast dLLM: First, let's calculate the total amount Janeth needs to return. She borrowed $2000 and promised to return an additional 10% of the amount.

$$\text{Additional amount} = 0.10 \times 2000 = 200$$

$$\text{Total amount to return} = 2000 + 200 = 2200$$

Next, let's calculate the total amount Janeth will pay over 12 months. She pays $165 per month.

$$\text{Total amount paid} = 165 \times 12 = 1880$$

Now, let's find the remaining balance after 12 months.

$$\text{Remaining balance} = \text{Total amount to return} - \text{Total amount paid}$$

$$\text{Remaining balance} = 2200 - 1880 = 320$$

So, Janeth's remaining balance after 12 months will be $320. $\boxed{320}$

Answer by HaloBlock: First, let's calculate the total amount Janeth needs to return. She borrowed $2000 and promised to return an additional 10% of the amount.

$$\text{Total amount} = 2000 + (0.10 \times 2000)$$

$$\text{Total amount} = 2000 + 200 = 2200$$

Next, let's calculate how much Janeth will pay over 12 months. She pays $165 per month.

$$\text{Total payment} = 165 \times 12$$

$$\text{Total payment} = 1980$$

Now, let's find the remaining balance after 12 months.

$$\text{Remaining balance} = \text{Total amount} - \text{Total payment}$$

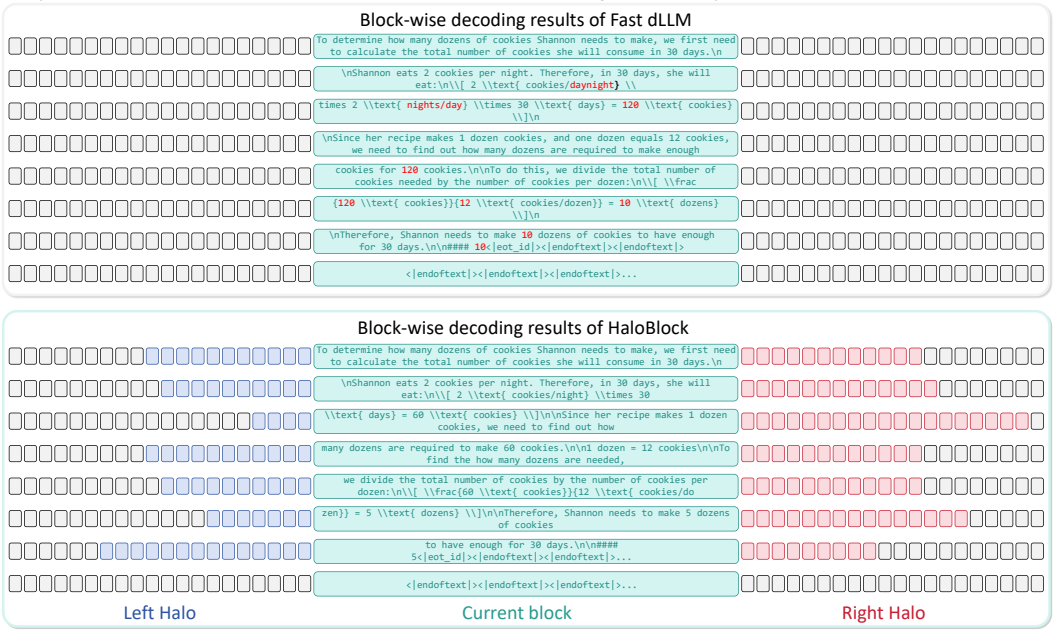
$$\text{Remaining balance} = 2200 - 1980 = 220$$

So, Janeth's remaining balance after 12 months will be:

220.

Figure 9: **Example 2**: Comparison of inference results between the baseline Fast dLLM and HaloBlock. We visualize the dynamic halo length in colors. Reasoning errors are highlighted in red.

Prompt: Shannon makes her own madeleine cookies and eats 2 a night as a treat. She wants to make enough cookies to last her for 30 days by storing them in the freezer. Her recipe makes 1 dozen madeleine cookies. How many dozens of cookies will she need to make so she has enough for 30 days?



Note: In the above figure, the reasoning errors are highlighted in red and the missing predictions are highlight in **bold black**.

Answer by Fast dLLM: To determine how many dozens of cookies Shannon needs to make, we first need to calculate the total number of cookies she will consume in 30 days.

Shannon eats 2 cookies per night. Therefore, in 30 days, she will eat:

$$2 \text{ cookies/daynight} \times 2 \text{ nights/day} \times 30 \text{ days} = 120 \text{ cookies}$$

Since her recipe makes 1 dozen cookies, and one dozen equals 12 cookies, we need to find out how many dozens are required to make enough cookies for 120 cookies.

To do this, we divide the total number of cookies needed by the number of cookies per dozen:

$$\frac{120 \text{ cookies}}{12 \text{ cookies/dozen}} = 10 \text{ dozens}$$

Therefore, Shannon needs to make 10 dozens of cookies to have enough for 30 days.

10

Answer by HaloBlock: To determine how many dozens of cookies Shannon needs to make, we first need to calculate the total number of cookies she will consume in 30 days.

Shannon eats 2 cookies per night. Therefore, in 30 days, she will eat:

$$2 \text{ cookies/night} \times 30 \text{ days} = 60 \text{ cookies}$$

Since her recipe makes 1 dozen cookies, we need to find out how many dozens are required to make 60 cookies.

1 dozen = 12 cookies

To find the how many dozens are needed, we divide the total number of cookies by the number of cookies per dozen:

$$\frac{60 \text{ cookies}}{12 \text{ cookies/dozen}} = 5 \text{ dozens}$$

Therefore, Shannon needs to make 5 dozens of cookies to have enough for 30 days.

5

Figure 10: **Example 3**: Comparison of inference results between the baseline Fast dLLM and HaloBlock. We visualize the dynamic halo length in colors. Reasoning errors are highlighted in red.