DEPCAP: ADAPTIVE BLOCK-WISE PARALLEL DECODING FOR EFFICIENT DIFFUSION LM INFERENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion language models (DLMs) offer a promising alternative to autoregressive language generation by enabling parallel decoding and global sequence refinement. To unlock this potential, DLM inference must balance generation quality and decoding speed. Recent block-wise DLM decoding methods improve this trade-off by performing diffusion-based decoding sequentially in blocks. However, existing block-wise methods often rely on signals that are not well aligned with two key decisions in DLM inference. For block partitioning, fixed schedules and current-step local signals do not explicitly capture cross-step prediction changes, while confidence-based parallel decoding does not explicitly consider interactions between token predictions. In this paper, we argue that block-wise DLM inference requires decision-matched signals: cross-step signals for block partitioning and token-level conflict signals for parallel decoding. Based on this view, we propose DepCap, a training-free framework that instantiates the cross-step signal as the influence of the last decoded block and uses it to adaptively determine the next block boundary. Within each block, DepCap combines token confidence with a pairwise conflict score to select tokens for parallel decoding, balancing decoding speed and generation quality. DepCap is plug-and-play across DLMs and compatible with existing block-wise KV-cache strategies. An information-theoretic analysis characterizes the deviation from additive last-block influence, motivating the proposed block-partitioning criterion. Experiments across four DLM backbones and four reasoning and coding benchmarks show favorable quality-speed trade-offs, with up to 5.63$\times$ speedup over vanilla decoding.

1 INTRODUCTION

Diffusion language models (DLMs) (Lou et al., 2024; Li et al., 2025b; Yu et al., 2025) have emerged as a promising alternative to autoregressive language generation because they can refine many token positions in parallel instead of decoding tokens strictly from left to right. Unlike autoregressive models, DLMs can use bidirectional attention during the generation process, allowing each position to be updated using both left and right context. This property makes DLMs appealing for flexible text generation, especially on tasks such as reasoning (Zhao et al., 2025; Huang et al., 2025) and coding (Gong et al., 2026), where decoding efficiency and rich contextual modeling are both important. Recent DLMs, including the Mercury (Labs et al., 2025), Dream (Ye et al., 2025) and LLaDA (Nie et al., 2025; Bie et al., 2025) families, further show that this paradigm is increasingly practical and promising. However, due to the multi-step denoising process and the need for global refinement during inference, DLMs inherently face a trade-off between generation quality and speed, making efficient inference a critical challenge for practical deployment under limited latency and computational budgets (Wu et al., 2026b; Ma et al., 2025).

To fully unlock the parallel decoding potential of DLMs, recent work has increasingly adopted block-wise decoding strategies (Nie et al., 2025; Wu et al., 2026b; Arriola et al., 2025) for inference. Instead of refining the entire unfinished sequence at every step, block-wise DLMs decode the output sequentially in blocks, which requires *two key inference-time decisions: how to determine the next block boundary and which tokens to decode in parallel within the current block*. This strategy significantly reduces the refinement scope at each iteration while allowing each decoded block to serve as clean context for subsequent blocks, thereby improving the practicality of DLM inference. It

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.

also enables subsequent acceleration techniques, including KV-cache mechanisms (Ma et al., 2025; Wu et al., 2026b; Song et al., 2026) tailored to block-wise DLM inference.

However, existing block-wise DLM inference methods still leave a substantial quality-speed gap. The main limitation is that the signals used for these two key inference decisions are not fully aligned with *two defining characteristics of DLM inference: the evolving nature of the denoising process and its parallel token-decoding behavior*. First, at the block level, most block-wise methods either use a fixed block size or determine the next boundary from signals available only at the current decoding step, rather than from signals that reflect the evolving multi-step denoising process of DLM inference. Second, at the token level, many methods rely on conservative confidence-based parallel decoding strategies. Such strategies treat candidate positions largely independently and fail to account for token conflicts introduced by parallel decoding: lowering the confidence threshold may improve decoding speed, but it can also decode conflicting tokens simultaneously, leading to degraded quality (Luo et al., 2026b). These observations motivate cross-step signals for block partitioning and pairwise predictive signals for parallel token selection.

Problem. How can block-wise DLM inference match each core decision with the signal required by the underlying inference behavior: cross-step evidence for block boundary selection and token-level conflict evidence for parallel decoding, thereby achieving a better quality-speed trade-off?

Contribution. Our key contribution is a decision-signal matching view of block-wise DLM inference: *each inference-time decision should be guided by the signal it naturally requires: a cross-step denoising signal for block partitioning and a token-level conflict signal for within-block parallel decoding.* Based on this view, we propose DepCap, a training-free framework for block-wise DLM inference. At the block level, DepCap goes beyond fixed schedules and current-step local cues by introducing a cross-step signal derived from the last decoded block. Specifically, the proposed dependency-guided adaptive block partitioning strategy (DepGA-Block) measures how strongly the last decoded block influences future positions and combines this signal with predictive uncertainty through a dependency score to determine the next block boundary. Within each chosen block, conflict-aware parallel decoding (CAP-Decoding) combines high-confidence token selection with pairwise conflict filtering to support more aggressive parallel decoding. DepCap is plug-and-play and compatible with existing block-wise cache strategies. An information-theoretic analysis characterizes the error in approximating joint last-block influence by per-position contributions, providing a theoretical motivation for the dependency score used in block partitioning.

Experiments show favorable quality-speed trade-offs across the LLaDA (Nie et al., 2025; Zhu et al., 2025a;b) and Dream (Ye et al., 2025) families on reasoning and coding benchmarks. On the two dense LLaDA backbones, DepCap attains an average 3.57$\times$ speedup over vanilla decoding while retaining or improving accuracy. For example, on MBPP with LLaDA-1.5, DepCap achieves a 5.63$\times$ speedup while improving accuracy by 7.4% relative.

The subsequent sections present the related work and preliminaries, describe the proposed DepCap method, show the empirical results, and conclude the paper.

2 Related Work

Diffusion language models. Diffusion language models (DLMs) (Li et al., 2025b; Yu et al., 2025) formulate text generation as an iterative denoising process rather than a strictly left-to-right prediction process. This paradigm originates from diffusion modeling for continuous domains (Ho et al., 2020) and has been extended to discrete text generation (Gong et al., 2023; Austin et al., 2021a). A key advantage of DLMs is their ability to perform parallel token updates and leverage bidirectional context during inference, enabling non-sequential generation that differs fundamentally from autoregressive generation. Recent work (Gong et al., 2025; Nie et al., 2025; Ye et al., 2025; Cheng et al., 2025) has further scaled DLMs to stronger language modeling settings, showing that diffusion-based generation is becoming increasingly practical, with models reaching 100B parameters (e.g., LLaDA 2.0 (Bie et al., 2025)). This non-sequential paradigm offers greater flexibility than autoregressive generation but introduces a quality-speed trade-off.

Block-wise decoding in DLMs. To improve the quality-speed trade-off of DLM inference, recent work adopts a block-wise decoding strategy (Arriola et al., 2025; Wu et al., 2026b; Nie et al., 2025; Wu et al., 2026a), where the sequence is decoded sequentially in blocks rather than repeatedly

refining unfinished positions at every step. Existing block-wise DLMs often rely on fixed block sizes. This fixed-size design does not account for variation across inputs or across stages of the same generation process. Therefore, more recent approaches attempt to dynamically adjust block size. AdaBlock (Lu et al., 2026) determines block boundaries by examining semantic structure, such as special tokens (e.g., "\n"), while Swordsman (Zhang et al., 2026) partitions blocks based on entropy changes across adjacent positions. A concurrent work, GeoBlock (Wan et al., 2026), infers block size from current-step attention-derived dependency geometry. These methods primarily use current-step signals for block partitioning, without explicitly using prediction changes induced by decoding the preceding block.

Efficient inference for DLMs. To further unlock the parallel generation potential of DLMs, existing work mainly focuses on decoding strategies and cache techniques. For decoding strategies, recent methods accelerate inference via heuristic-based parallel decoding (Wu et al., 2026b; Luo et al., 2026b; Zhang et al., 2026; Cao et al., 2026), learned parallel decoding (Bao et al., 2026; Chen et al., 2026), denoising-process redesign (Wang et al., 2026), and speculative decoding (Hu et al., 2026; Israel et al., 2025; Li et al., 2025a). Cache techniques (Wu et al., 2026b; Ma et al., 2025; Liu et al., 2025; Jiang et al., 2026) form the other direction, and their core idea is typically to refresh cached states at selected intervals based on the similarity of hidden states across adjacent decoding steps. While these approaches show that substantial DLM inference acceleration is possible, achieving an effective balance between generation quality and decoding speed remains a significant challenge.

3 PRELIMINARIES

Diffusion language model inference. Diffusion language models (DLMs) considered here adopt a masked language modeling formulation. Let $\mathcal{V}$ denote the token vocabulary, including the special token `[MASK]`. Given a prompt x, let $\boldsymbol{y} = (y_1, \ldots, y_N) \in \mathcal{V}^N$ denote the target sequence to be generated. In inference, the model takes as input the prompt x followed by the current target sequence $\boldsymbol{y}$. At the beginning of inference, this sequence is initialized as $\boldsymbol{y}^T = (\texttt{[MASK]}, \ldots, \texttt{[MASK]})$. At denoising step t, the current state of the target sequence is $\boldsymbol{y}^t = (y_1^t, \ldots, y_N^t) \in \mathcal{V}^N$. The masked positions at step t are $M^t = \{i : y_i^t = \texttt{[MASK]}\}$. At each step, the model first predicts token distributions for positions in M^t, and then a decoding strategy chooses a subset $S^t \subseteq M^t$ to decode:

$$y_i^{t-1} = \begin{cases} \hat{y}_i^t, & i \in S^t, \\ \texttt{[MASK]}, & i \in M^t \setminus S^t, \\ y_i^t, & i \notin M^t, \end{cases} \tag{1}$$

where $\hat{y}_i^t$ is the predicted token at position i. Repeating this iterative decoding process gradually removes mask tokens from $\boldsymbol{y}^t$ until all masked positions have been decoded, and the final sequence satisfies $\boldsymbol{y}^0 \in (\mathcal{V} \setminus \{\texttt{[MASK]}\})^N$. Unlike autoregressive decoding, each denoising step can update multiple positions in parallel while using bidirectional context.

Block-wise decoding. To better balance generation quality and speed, recent DLM inference methods often partition the target sequence into a sequence of non-overlapping blocks $\mathcal{B} = \{B_1, \ldots, B_K\}$. During inference, blocks are decoded sequentially: block B_{k+1} starts only after block B_k has been decoded. For brevity, we define the current decoding block at step t as $B^t \in \mathcal{B}$. Then block-wise decoding restricts the decoding subset to $S^t \subseteq M^t \cap B^t$. Equivalently, the sequence update is restricted to masked positions within the current block:

$$y_i^{t-1} = \begin{cases} \hat{y}_i^t, & i \in S^t, \\ \texttt{[MASK]}, & i \in (M^t \cap B^t) \setminus S^t, \\ y_i^t, & i \notin M^t \cap B^t, \end{cases} \tag{2}$$

which means that only positions in the current block are allowed to decode at step t.

Two decisions in block-wise inference. Block-wise decoding introduces two inference decisions. The first is the *block-boundary decision*: given the already decoded context, how large should the next block be? The second is the *within-block decoding decision*: within the chosen block, which positions should be decoded together at the current denoising step? Existing block-wise DLM inference methods mainly differ in how they make these two decisions.

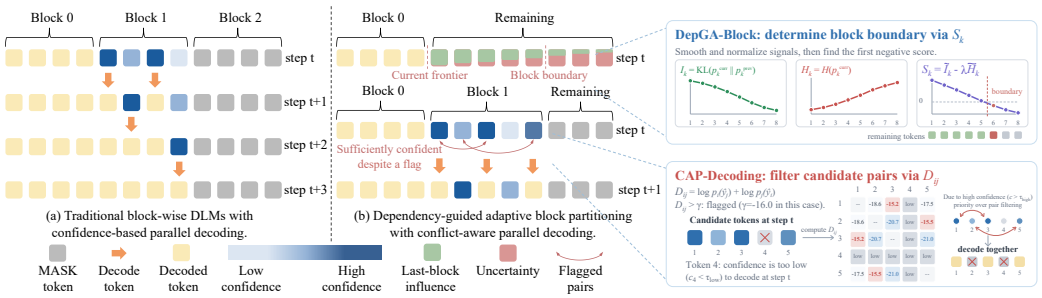


Figure 1: The framework of DepCap. (a) Traditional block-wise DLM inference uses a fixed block size and confidence-only parallel decoding. (b) DepCap chooses the next block boundary with DepGA-Block based on last-block influence and predictive uncertainty, and uses CAP-Decoding to select positions for parallel decoding. Darker color indicates higher confidence, and red bidirectional arrows denote pairs flagged by the conflict score. Green and red in DepGA-Block indicate last-block influence and predictive uncertainty. The first negative dependency score determines the next block boundary within the configured minimum and maximum block lengths.

4 The DepCap Framework

This section presents DepCap, a training-free inference framework for block-wise diffusion language models (DLMs). We first give an overview of DepCap, then describe the dependency-guided adaptive block partitioning strategy (DepGA-Block) and the conflict-aware parallel decoding strategy (CAP-Decoding), and finally analyze the dependency score used for block partitioning.

4.1 Overview

DepCap is built around a simple principle: *each inference-time decision in block-wise DLM decoding should use the signal that naturally matches it*. At each stage, DepCap makes two decisions: how far the next block should extend, and which positions inside that block to decode in parallel. For the first decision, DepGA-Block goes beyond fixed schedules and current-step local cues by using a cross-step signal derived from the last decoded block to guide block expansion. For the second, CAP-Decoding filters candidates using pairwise conflict scores in addition to token confidence. Because both modules operate only on inference-time predictive signals, DepCap can be added on top of existing block-wise DLM inference processes without retraining, and is compatible with existing block-wise cache techniques. Figure 1 illustrates the overall DepCap framework.

4.2 Dependency-Guided Adaptive Block Partitioning Strategy

The purpose of the dependency-guided adaptive block partitioning strategy (DepGA-Block) is to adapt block boundaries to the evolving DLM generation process. Existing methods typically determine block boundaries either with a fixed block (Wu et al., 2026b) or from current-step signals, such as semantic structure (Lu et al., 2026), entropy (Zhang et al., 2026), or attention (Wan et al., 2026). In contrast, DepGA-Block follows the multi-step denoising behavior of DLM inference and introduces a cross-step signal derived from the last decoded block to measure how newly decoded context changes the model's predictions for future positions.

Specifically, suppose the last decoded block B has just been added to the decoded context. For each candidate future position k, let p_k^{prev} denote the predictive distribution before decoding B, and let p_k^{curr} denote the predictive distribution after decoding B. In implementation, before decoding each block, DepGA-Block caches the predictive distributions over the candidate window as p_k^{prev} for selecting the following block, avoiding an extra model forward pass. DepGA-Block measures the influence of the last decoded block by the KL divergence between these two predictive distributions:

$$I_k = \mathrm{KL}\left(p_k^{\text{curr}} \,\middle\|\, p_k^{\text{prev}}\right), \tag{3}$$

which we call *last-block influence*. A larger I_k indicates a stronger change in the marginal prediction at position k after decoding the last block. DepGA-Block combines this signal with predictive

uncertainty to determine the next block boundary. To complement this influence signal, DepGA-Block measures predictive uncertainty by the Shannon entropy of the predictive distribution: $H_k = \mathrm{H}(p_k^{\text{curr}})$. Large entropy indicates that the position remains uncertain even after the last block has been decoded. Together, I_k and H_k describe how strongly the predictions change and how uncertain they remain after decoding the last block.

In practice, both signals are smoothed and normalized within a local candidate window of size $L_{\text{window}} = \min(L_{\max}, L_{\text{remain}})$, where L_{remain} denotes the number of remaining masked tokens after the current decoding frontier. This yields the normalized signals $\tilde{I}_k$ and $\tilde{H}_k$. DepGA-Block then defines the *dependency score* to determine the block size:

$$S_k = \tilde{I}_k - \lambda \tilde{H}_k, \tag{4}$$

where $\lambda > 0$ controls the relative weight of locally normalized uncertainty. Starting from the current frontier, DepCap finds the first position where $S_k < 0$, meaning that weighted uncertainty exceeds last-block influence in the normalized score. The next block ends immediately before that position, or spans the candidate window if no such position exists. The final length L is clipped to $[\min(L_{\min}, L_{\text{window}}), L_{\text{window}}]$. The first block uses $L = \min(L_{\min}, L_{\text{remain}})$ as a conservative cold start. Section 4.4 provides the analysis, and Appendix A details signal computation.

All signals used here are available from the model's predictive distributions during inference. The chosen boundary remains fixed while the block is decoded, whereas CAP-Decoding recomputes token selection at each denoising step.

4.3 Conflict-Aware Parallel Decoding Strategy

For within-block parallel decoding, the key question is not only which positions are confident, but also how their predictions interact. Traditional confidence-based decoding strategies often employ conservative thresholds to preserve generation quality, limiting within-block parallelism. CAP-Decoding combines high-confidence token selection with pairwise conflict filtering to support more aggressive parallel decoding, as summarized in Algorithm 1.

Specifically, once the next block B^t has been chosen, CAP-Decoding decides which masked positions within that block can be decoded simultaneously. Given the token confidence $c_i = \max_v p_i(v)$ at position i, we define the candidate pool in the current block as:

$$\mathcal{C} = \{i : i \in M^t \cap B^t,\ c_i \geq \tau_{\text{low}}\} \tag{5}$$

where M^t denotes the set of still masked tokens in the sequence. Confidence filters potential candidates but does not describe interactions between their predictions.

To select candidates for parallel decoding, CAP-Decoding employs a lightweight pairwise conflict score. For distinct positions $i, j \in \mathcal{C}$, the score is computed as:

$$D_{ij} = \log p_i(\hat{y}_j) + \log p_j(\hat{y}_i), \tag{6}$$

where $\hat{y}_i$ and $\hat{y}_j$ are the current token predictions at positions i and j, respectively. A high D_{ij} indicates strong cross-support between the current predictions at the two positions. CAP-Decoding uses this score as an inference-time proxy for potential token conflicts. Pairs satisfying $D_{ij} > \gamma$ are flagged for conflict filtering during subset selection.

CAP-Decoding selects the parallel subset in two phases. First, it selects all candidates satisfying $c_i \geq \tau_{\text{high}}$ and removes them and their conflicting neighbors from $\mathcal{C}$. Next, it repeatedly selects the highest-confidence remaining candidate and removes that candidate and its conflicting neighbors. High-confidence candidates take priority over conflict filtering. If no token is selected, CAP-Decoding chooses the highest-confidence masked token to ensure progress.

This procedure allows a broader candidate pool than conservative confidence-only decoding while using pairwise scores to guide parallel token selection.

4.4 Analysis of the Dependency Score

We interpret last-block influence using an idealized joint distribution $p(B, Z_{1:L} \mid c)$ over finite token sequences, with decoded context c fixed. Here B denotes the token content of the preceding block,

Algorithm 1 Conflict-Aware Parallel Decoding Strategy (CAP-Decoding)

Require: Current block B^t, masked-position set M^t, predictive distributions $\{p_i\}_{i\in B^t}$, confidence thresholds $\tau_{\text{low}}, \tau_{\text{high}}$, and conflict threshold γ
Ensure: Selected subset $\mathcal{S} \subseteq M^t \cap B^t$ for parallel decoding
Compute token confidence $c_i = \max_v p_i(v)$ for each candidate $i \in M^t \cap B^t$
Build candidate pool $\mathcal{C} = \{i : i \in M^t \cap B^t,\ c_i \geq \tau_{\text{low}}\}$
for each $(i, j) \in \mathcal{C} \times \mathcal{C}$ with $i \neq j$ **do**
    Let $\hat{y}_i = \arg\max_v p_i(v)$ and $\hat{y}_j = \arg\max_v p_j(v)$
    Compute conflict score $D_{ij} = \log p_i(\hat{y}_j) + \log p_j(\hat{y}_i)$
end for
/* Phase 1: High-confidence priority */
Initialize $\mathcal{S} = \{i \in \mathcal{C} : c_i \geq \tau_{\text{high}}\}$
Remove $\mathcal{S}$ and all candidates that conflict with any candidate in $\mathcal{S}$ from $\mathcal{C}$
/* Phase 2: Greedy completion */
while $\mathcal{C} \neq \emptyset$ **do**
    Select $i = \arg\max_{j\in\mathcal{C}} c_j$
    Add candidate i to $\mathcal{S}$
    Remove candidate i and the candidates that conflict with i from $\mathcal{C}$
end while
if $\mathcal{S} = \emptyset$ **then**
    Add the candidate $i = \arg\max_{i\in M^t\cap B^t} c_i$ to $\mathcal{S}$
end if
return $\mathcal{S}$

and $Z_{1:L}$ denotes the candidate next block. All conditionals below derive from this joint. For a realized block $B = b$, define $J_k(b) = \mathrm{KL}(p(Z_k \mid c, b) \| p(Z_k \mid c))$. Averaging over the preceding block B gives the following mutual-information identity:

$$I_k^{\text{exp}} = \mathbb{E}_B[J_k(B)] = \mathrm{I}(Z_k; B \mid c). \tag{7}$$

This relates changes in marginal predictions to information supplied by the last block. The joint influence $\mathcal{I}(L) = \mathrm{I}(Z_{1:L}; B \mid c)$ satisfies the exact decomposition

$$\begin{aligned} \mathcal{I}(L) &= \sum_{k=1}^{L} I_k^{\text{exp}} - \varepsilon(L), \\ \varepsilon(L) &= \mathrm{TC}(Z_{1:L} \mid c) - \mathrm{TC}(Z_{1:L} \mid c, B), \end{aligned} \tag{8}$$

where TC denotes total correlation and the second term averages over B. Thus, additivity is accurate when conditioning on the last block changes total correlation only slightly, even if future token positions within the candidate generation window remain statistically dependent.

Let $\bar{h}_k = \mathrm{H}(Z_k \mid c, B)$ denote expected predictive uncertainty. An idealized design criterion compares joint influence with cumulative uncertainty, $\mathcal{I}(L) \geq \lambda \sum_k \bar{h}_k$. Writing $A(L) = \sum_k (I_k^{\text{exp}} - \lambda \bar{h}_k)$, this criterion becomes $A(L) \geq \varepsilon(L)$. Its additive approximation $A(L) \geq 0$ gives the same decision whenever $|\varepsilon(L)| < |A(L)|$. Appendix B proves these statements.

The resulting influence-minus-uncertainty form motivates Equation (4). DepGA-Block turns this motivation into an efficient local boundary decision using cross-step model predictions, local normalization, and a first-negative-score rule.

5 Experiments

To evaluate the quality-speed trade-off of DepCap, we conduct experiments across representative DLM backbones and benchmarks. The experiments address the following questions:

- **Q1:** Does DepCap improve the overall quality-speed trade-off of block-wise DLM inference?
- **Q2:** Does DepCap remain effective when combined with existing block-wise cache techniques?
- **Q3:** How much do DepGA-Block and CAP-Decoding each contribute to the final gains?

Table 1: Main results on three DLM backbones and four benchmarks. We report task accuracy (Acc), throughput in tokens per second (TPS), and number of function evaluations (NFE) under different block partitioning and decoding strategies. "Vanilla" denotes decoding only the highest confidence token within the current block at each step and "Confidence" denotes the parallel decoding strategy of Fast-dLLM. Best results are **bolded**, and second-best are underlined.

Parallel Decode	Block	GSM8K (5-shot)			Math-500 (4-shot)			MBPP (3-shot)			HumanEval (0-shot)		
		TPS ⇑	NFE ⇓	Acc ⇑	TPS ⇑	NFE ⇓	Acc ⇑	TPS ⇑	NFE ⇓	Acc ⇑	TPS ⇑	NFE ⇓	Acc ⇑
LLaDA-8B-Instruct													
Vanilla	$\|B_0\| = 32$	$6.0_{\times 1.00}$	256.0	77.6	$7.9_{\times 1.00}$	256.0	39.2	$5.4_{\times 1.00}$	256.0	40.8	$16.7_{\times 1.00}$	256.0	42.7
Confidence	$\|B_0\| = 16$	$18.3_{\times 3.05}$	84.1	78.3	$19.7_{\times 2.49}$	104.4	40.2	$18.3_{\times 3.39}$	75.9	41.2	$51.0_{\times 3.05}$	86.4	**43.3**
	$\|B_0\| = 32$	$19.6_{\times 3.27}$	78.6	77.3	$20.7_{\times 2.62}$	99.1	39.6	$19.7_{\times 3.65}$	70.1	41.0	$56.5_{\times 3.38}$	77.5	**43.3**
	$\|B_0\| = 64$	$20.1_{\times 3.35}$	75.6	77.5	$21.1_{\times 2.67}$	97.0	37.8	$19.9_{\times 3.69}$	68.2	40.8	$57.2_{\times 3.43}$	76.4	41.5
Swordsman		$20.6_{\times 3.43}$	75.2	77.8	$20.9_{\times 2.65}$	98.4	33.0	**21.8**$_{\times 4.04}$	**65.2**	**41.4**	$59.3_{\times 3.55}$	72.1	40.9
Confidence	AdaBlock	$18.4_{\times 3.07}$	83.9	77.9	$19.8_{\times 2.51}$	103.5	40.0	$19.4_{\times 3.59}$	76.2	40.4	$50.4_{\times 3.02}$	86.2	40.9
	DepGA-Block	$18.7_{\times 3.12}$	81.4	**79.2**	$19.9_{\times 2.52}$	101.8	**40.4**	$18.8_{\times 3.48}$	73.6	41.0	$53.4_{\times 3.20}$	80.6	42.7
CAP-Decoding	$\|B_0\| = 32$	**21.6**$_{\times 3.60}$	**71.5**	77.3	**22.8**$_{\times 2.89}$	**90.9**	39.2	$21.2_{\times 3.93}$	65.7	40.6	**60.4**$_{\times 3.62}$	**72.0**	**43.3**
	DepGA-Block	$21.4_{\times 3.57}$	74.8	78.8	$22.2_{\times 2.81}$	94.8	39.8	$20.8_{\times 3.85}$	68.6	40.8	$59.8_{\times 3.58}$	74.8	42.7
Dream-v0-base-7B													
Vanilla	$\|B_0\| = 32$	$7.8_{\times 1.00}$	256.0	**75.1**	$9.7_{\times 1.00}$	256.0	**42.2**	$10.2_{\times 1.00}$	256.0	50.8	$20.7_{\times 1.00}$	256.0	49.4
Confidence	$\|B_0\| = 16$	**13.7**$_{\times 1.76}$	148.1	74.7	$23.7_{\times 2.44}$	105.5	40.8	$27.7_{\times 2.72}$	94.1	53.2	$38.8_{\times 1.87}$	137.0	**55.5**
	$\|B_0\| = 32$	$13.2_{\times 1.69}$	152.4	74.5	$24.4_{\times 2.52}$	102.3	40.4	$28.6_{\times 2.80}$	90.8	51.8	$39.0_{\times 1.88}$	135.3	53.0
	$\|B_0\| = 64$	$12.8_{\times 1.64}$	157.9	74.6	**24.9**$_{\times 2.57}$	99.5	41.0	$28.9_{\times 2.83}$	89.1	52.2	$39.4_{\times 1.90}$	131.8	53.7
Swordsman		$12.5_{\times 1.60}$	158.3	73.0	$24.1_{\times 2.48}$	100.1	40.8	$27.7_{\times 2.72}$	92.2	**53.4**	$38.4_{\times 1.86}$	132.7	52.4
Confidence	AdaBlock	$13.1_{\times 1.68}$	154.9	**75.1**	$24.7_{\times 2.55}$	101.7	39.8	$28.2_{\times 2.76}$	93.7	52.6	$38.6_{\times 1.86}$	137.9	53.7
	DepGA-Block	$13.5_{\times 1.73}$	149.0	75.0	$24.0_{\times 2.47}$	103.2	41.0	$28.7_{\times 2.81}$	90.1	52.6	$39.5_{\times 1.91}$	132.5	50.6
CAP-Decoding	$\|B_0\| = 32$	$12.7_{\times 1.63}$	149.9	74.1	$24.7_{\times 2.55}$	**95.4**	40.4	$29.0_{\times 2.84}$	85.2	52.4	$39.8_{\times 1.92}$	127.3	50.6
	DepGA-Block	$13.0_{\times 1.67}$	**146.6**	74.8	$24.4_{\times 2.52}$	97.2	40.8	**29.2**$_{\times 2.86}$	**84.6**	52.8	**40.6**$_{\times 1.96}$	**125.3**	50.6
LLaDA-1.5													
Vanilla	$\|B_0\| = 32$	$5.6_{\times 1.00}$	256.0	81.1	$7.1_{\times 1.00}$	256.0	39.8	$1.9_{\times 1.00}$	256.0	37.8	$5.9_{\times 1.00}$	256.0	41.5
Confidence	$\|B_0\| = 16$	$17.5_{\times 3.12}$	81.9	81.4	$17.8_{\times 2.51}$	101.0	40.0	$10.4_{\times 5.47}$	48.7	40.2	$14.2_{\times 2.41}$	106.9	40.9
	$\|B_0\| = 32$	$18.9_{\times 3.38}$	75.5	81.2	$18.9_{\times 2.66}$	95.2	39.4	$11.6_{\times 6.11}$	42.1	37.6	$15.3_{\times 2.59}$	99.6	40.2
	$\|B_0\| = 64$	$19.5_{\times 3.48}$	72.8	80.7	$19.3_{\times 2.72}$	91.7	38.0	$9.2_{\times 4.84}$	40.1	25.4	$16.1_{\times 2.73}$	93.4	**42.1**
Swordsman		$19.4_{\times 3.46}$	74.7	73.2	$18.1_{\times 2.55}$	101.0	31.8	**12.8**$_{\times 6.74}$	**37.4**	39.2	**16.6**$_{\times 2.81}$	**87.3**	40.2
Confidence	AdaBlock	$17.8_{\times 3.18}$	80.1	82.1	$18.5_{\times 2.61}$	97.6	38.8	$9.8_{\times 5.16}$	51.8	36.4	$14.1_{\times 2.39}$	110.9	37.8
	DepGA-Block	$18.1_{\times 3.23}$	80.1	80.6	$18.1_{\times 2.55}$	100.0	41.6	$10.3_{\times 5.42}$	50.1	40.0	$14.8_{\times 2.51}$	105.6	39.6
CAP-Decoding	$\|B_0\| = 32$	**20.6**$_{\times 3.68}$	**69.1**	82.1	**20.2**$_{\times 2.85}$	**88.8**	40.0	$12.2_{\times 6.42}$	40.6	39.0	$16.4_{\times 2.78}$	93.0	**42.1**
	DepGA-Block	$20.1_{\times 3.59}$	73.9	**82.2**	$19.7_{\times 2.77}$	94.1	**42.2**	$10.7_{\times 5.63}$	48.9	**40.6**	$16.2_{\times 2.75}$	98.5	41.5

Q4: How does DepCap behave under different hyperparameter choices?

Our code is available at `https://anonymous.4open.science/r/DepCap-434C`.

5.1 Experimental Settings

We conduct comprehensive evaluations on four representative DLM backbones, specifically LLaDA-8B-Instruct (Nie et al., 2025), Dream-v0-base-7B (Ye et al., 2025), LLaDA-1.5 (Zhu et al., 2025a) and an MoE-based backbone, LLaDA-MoE-7B-A1B-Instruct (Zhu et al., 2025b), to validate the effectiveness of the proposed DepCap. Additionally, we compare DepCap against the established block-wise DLM framework Fast-dLLM (Wu et al., 2026b), the adaptive block partition method AdaBlock (Lu et al., 2026), and Swordsman (Zhang et al., 2026), a block-wise method that combines adaptive block partitioning with a dynamic decoding strategy.

Hyperparameter settings. Unless otherwise specified, all methods use a generation length of $L_{\text{gen}} = 256$. In DepCap, DepGA-Block uses $\lambda = 1.2$, $L_{\min} = 8$, and $L_{\max} = 128$, while CAP-Decoding uses $\tau_{\text{low}} = 0.8$, $\tau_{\text{high}} = 0.95$, and $\gamma = -16.0$. We choose τ_{low} once on LLaDA-8B-Instruct/GSM8K and keep the same DepCap configuration across all backbones and tasks. Fast-dLLM uses confidence threshold 0.9 and fixed block sizes $|B_0| \in \{16, 32, 64\}$. AdaBlock uses its official default configuration, and Swordsman uses $\tau_{\text{init}} = 0.9$ and $\tau_{\text{low}} = 0.1$.

Benchmarks and metrics. We evaluate all methods on GSM8K (5-shot) (Cobbe et al., 2021) and Math-500 (4-shot) (Lightman et al., 2024) for mathematical reasoning, and MBPP (3-shot) (Austin et al., 2021b) and HumanEval (0-shot) (Chen et al., 2021) for code generation. We report task accuracy (Acc), tokens per second (TPS), and average number of function evaluations per example (NFE). All methods use temperature-zero decoding with batch size 1 on a single NVIDIA A100 GPU under the same `lm-eval` framework. TPS measures generated tokens per second over the full generation loop, including decoding overhead. Appendix C gives the detailed evaluation protocol.

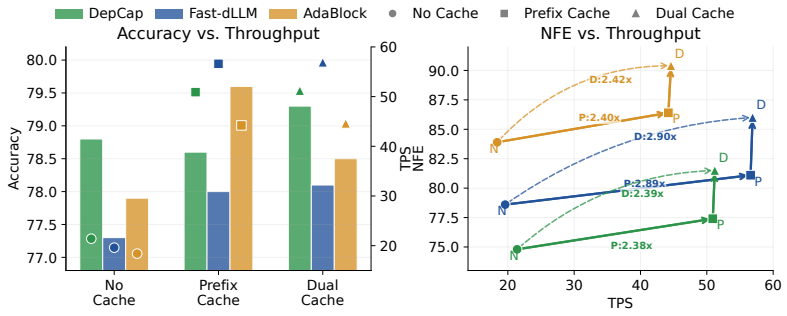


Figure 2: Cache-compatible results under No Cache, Prefix Cache, and Dual Cache. Left: accuracy versus throughput. Right: NFE versus throughput.

5.2 Main Results

To answer Q1. Table 1 reports results on LLaDA-8B-Instruct, Dream-v0-base-7B, and LLaDA-1.5 across the four benchmarks. Results for LLaDA-MoE-7B-A1B-Instruct can be found in Appendix D due to space limitations. A first observation is that the effect of fixed block size depends on the backbone and task. Larger blocks often reduce NFE but can lower accuracy, whereas smaller blocks restrict parallelism. This motivates adapting block boundaries to the generation process instead of relying on a fixed schedule. Appendix E illustrates the block boundaries and within-block decoding trajectories of DepCap and Fast-dLLM on individual GSM8K examples.

On the two dense LLaDA backbones, DepCap retains or improves accuracy over vanilla decoding across all four tasks, with an average 3.57× speedup. On MBPP with LLaDA-1.5, it achieves a 5.63× speedup while improving accuracy by 7.4% relative. Compared with AdaBlock and Swordsman, DepCap generally achieves higher accuracy on LLaDA reasoning tasks with competitive throughput. On Dream, acceleration comes with improved accuracy on code generation and modest reductions on mathematical reasoning relative to vanilla decoding. The results support DepCap's favorable quality-speed trade-off, while showing that the quality impact of acceleration depends on both the model family and the reasoning or code-generation task.

NFE and TPS capture different aspects of efficiency. NFE counts model evaluations, whereas TPS also reflects the cost of each evaluation and the decoding policy. We therefore interpret accuracy and throughput together when comparing the quality-speed trade-offs of inference strategies.

Runtime and repeatability. On LLaDA-8B-Instruct, model forward dominates measured GPU time and repeated runs show little TPS variation. Appendix F reports profiles and repeat statistics.

To answer Q2. We evaluate DepCap with block-wise cache techniques on LLaDA-8B-Instruct using GSM8K. We compare DepCap with Fast-dLLM (Wu et al., 2026b) and AdaBlock (Lu et al., 2026) under three cache settings: No Cache, Prefix Cache (Wu et al., 2026b) and Dual Cache (Wu et al., 2026b). Figure 2 reports accuracy, throughput, and NFE for these configurations.

Both Prefix and Dual Cache substantially improve throughput for all three methods, while accuracy changes remain small. Among the cache-enabled settings, Fast-dLLM achieves the highest throughput, whereas DepCap with Dual Cache reaches 79.3 accuracy with 51.2 TPS and lower NFE than the two cache-enabled AdaBlock variants. Caching reduces repeated model computation while retaining DepCap's adaptive decoding policy. These results show that DepCap combines naturally with existing block-wise cache techniques.

5.3 Ablation Study

To answer Q3. To analyze the role of each component, we compare "Confidence + DepGA-Block", "CAP-Decoding + Fixed Block ($|B_0| = 32$)", and the full "DepGA-Block + CAP-Decoding" method in Table 1. Vanilla serves as the basic decoding reference. For component analysis, we also compare each variant with confidence decoding at a fixed block size of 32.

The ablations show different effects of the two components. Under confidence-based decoding, DepGA-Block often improves accuracy relative to fixed partitioning. CAP-Decoding generally increases throughput under fixed block partitioning, while adding DepGA-Block often improves accuracy relative to fixed-block CAP-Decoding. Together, these results support combining adaptive block partitioning with conflict-aware parallel decoding to balance generation quality and decoding efficiency. Appendix G summarizes the cross-setting comparisons.

5.4 Hyperparameter Analysis

To answer Q4. We examine how hyperparameters affect decoding behavior and the quality-speed trade-off, varying one parameter at a time while keeping the others fixed. We analyze all five parameters on LLaDA-8B-Instruct/GSM8K and extend the λ and τ_{low} sweeps to LLaDA-8B-Instruct and Dream-v0-base-7B on Math-500 and HumanEval. Appendix H reports the complete sweeps and long-generation comparisons.

Generation length L_{gen}. Longer generations raise NFE but do not consistently improve accuracy or TPS. Comparisons at $L_{\text{gen}} = 1024$ extend this analysis to Math-500 and HumanEval. At this length, DepCap achieves the highest TPS among the four methods on HumanEval and balances accuracy and throughput between Fast-dLLM and AdaBlock on Math-500.

Maximum block size $L_{\max}$. A larger $L_{\max}$ raises TPS and lowers NFE until the gains saturate near the default. Further increasing the cap leaves NFE unchanged, consistent with adaptive boundaries rather than the cap alone determining the actual block lengths.

Uncertainty weight λ. Across the tested backbones and tasks, varying λ changes NFE and TPS only modestly, while accuracy shows no consistent monotonic trend. Since uncertainty is centered during normalization, increasing λ can raise or lower a position's score. Thus, λ adjusts the boundary criterion rather than imposing a fixed block length.

Confidence threshold τ_{low}. Stricter candidate filtering increases NFE and generally lowers TPS on LLaDA, with task-dependent accuracy changes. On Dream, accuracy and NFE remain unchanged on Math-500 and HumanEval over the tested range, showing backbone-dependent threshold sensitivity.

Conflict threshold γ. The conflict threshold γ controls filtering within the candidate pool. A looser threshold improves throughput, whereas more conservative filtering increases NFE. It adjusts parallelism after confidence filtering, while accuracy varies non-monotonically.

6 Conclusion and Discussion

Conclusion. This paper revisits block-wise DLM inference from a decision-signal matching perspective. We argue that its two key inference-time decisions require different signals: cross-step denoising signals for adaptive block partitioning and token-level conflict signals for within-block parallel decoding. Based on this view, we present DepCap, a training-free framework that instantiates these signals through dependency-guided adaptive block partitioning and conflict-aware parallel decoding. Our information-theoretic analysis characterizes the additive approximation underlying the dependency score, and experiments across multiple DLM backbones and reasoning and coding benchmarks show favorable quality-speed trade-offs while confirming compatibility with existing block-wise cache techniques.

Discussion. The current work focuses on block-wise DLM inference. A related and equally important direction is sliding-window diffusion decoding (Shu et al., 2026; Luo et al., 2026a; Nguyen-Tri et al., 2026), where adapting the window size by incorporating information from the historical multi-step denoising process is itself an interesting problem. The theoretical analysis characterizes the additive approximation, while stronger guarantees for the implemented block-selection rule remain an open direction. Further directions include broader task coverage and efficient batched inference for inputs with different adaptive block boundaries.

AI USE STATEMENT

Generative AI tools assist with manuscript drafting and language revision, helping improve grammar, clarity, and consistency of expression. The authors take responsibility for the final content, including the accuracy of technical descriptions and the interpretation of experimental results.

ETHICS STATEMENT

This work studies training-free inference acceleration for diffusion language models using publicly available pretrained models and established mathematical reasoning and code-generation benchmarks. It does not involve human-subject experiments or the collection of new personal data. Deployment of accelerated models should retain appropriate safeguards for generated text and code.

REPRODUCIBILITY STATEMENT

Sections 4 and 5 describe the DepCap framework and experimental settings. Appendix A details the dependency-score implementation, while Appendix C specifies the evaluation protocol, shared configuration, and metric definitions. Appendices F and H provide runtime profiles, repeated-run statistics, hyperparameter sweeps, and long-generation results. Experiments use publicly available model checkpoints and benchmarks under the same `lm-eval` framework. The anonymous implementation is linked in Section 5.

REFERENCES

Marianne Arriola, Aaron Gokaslan, Justin T. Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sekhar Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. In *Proceedings of the 13th International Conference on Learning Representations*, Singapore, Singapore, 2025.

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

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021b.

Wenrui Bao, Zhiben Chen, Dan Xu, and Yuzhang Shang. Learning to parallel: Accelerating diffusion large language models via learnable parallel decoding. In *Proceedings of the 14th International Conference on Learning Representations*, Rio de Janeiro, Brazil, 2026.

Tiwei Bie, Maosong Cao, Kun Chen, Lun Du, Mingliang Gong, Zhuochen Gong, Yanmei Gu, Jiaqi Hu, Zenan Huang, Zhenzhong Lan, Chengxi Li, Chongxuan Li, Jianguo Li, Zehuan Li, Huabin Liu, Lin Liu, Guoshan Lu, Xiaocheng Lu, Yuxin Ma, Jianfeng Tan, Lanning Wei, Ji-Rong Wen, Yipeng Xing, Xiaolu Zhang, Junbo Zhao, Da Zheng, Jun Zhou, Junlin Zhou, Zhanchao Zhou, Liwang Zhu, and Yihong Zhuang. LLaDA2.0: Scaling up diffusion language models to 100B. *CoRR*, abs/2512.15745, 2025.

Mingyu Cao, Alvaro H. C. Correia, Christos Louizos, Shiwei Liu, and Lu Yin. Search or accelerate: Confidence-switched position beam search for diffusion language models. *CoRR*, abs/2602.10953, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa,

Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

Zigeng Chen, Gongfan Fang, Xinyin Ma, Ruonan Yu, and Xinchao Wang. dParallel: Learnable parallel decoding for dLLMs. In *Proceedings of the 14th International Conference on Learning Representations*, Rio de Janeiro, Brazil, 2026.

Shuang Cheng, Yihan Bian, Dawei Liu, Linfeng Zhang, Qian Yao, Zhongbo Tian, Wenhai Wang, Qipeng Guo, Kai Chen, Biqing Qi, and Bowen Zhou. SDAR: A synergistic diffusion-autoregression paradigm for scalable sequence generation. *CoRR*, abs/2510.06303, 2025.

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

Shansan Gong, Mukai Li, Jiangtao Feng, Zhiyong Wu, and Lingpeng Kong. DiffuSeq: Sequence to sequence text generation with diffusion models. In *Proceedings of the 11th International Conference on Learning Representations*, Kigali, Rwanda, 2023.

Shansan Gong, Shivam Agarwal, Yizhe Zhang, Jiacheng Ye, Lin Zheng, Mukai Li, Chenxin An, Peilin Zhao, Wei Bi, Jiawei Han, Hao Peng, and Lingpeng Kong. Scaling diffusion language models via adaptation from autoregressive models. In *Proceedings of the 13th International Conference on Learning Representations*, Singapore, Singapore, 2025.

Shansan Gong, Ruixiang Zhang, Huangjie Zheng, Jiatao Gu, Navdeep Jaitly, Lingpeng Kong, and Yizhe Zhang. DiffuCoder: Understanding and improving masked diffusion models for code generation. In *Proceedings of the 14th International Conference on Learning Representations*, Rio de Janeiro, Brazil, 2026.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems 33*, Virtual Event, 2020.

Zhanqiu Hu, Jian Meng, Yash Akhauri, Mohamed S. Abdelfattah, Jae-sun Seo, Zhiru Zhang, and Udit Gupta. FlashDLM: Accelerating diffusion language model inference via efficient KV caching and guided diffusion. In *Proceedings of the 14th International Conference on Learning Representations*, Rio de Janeiro, Brazil, 2026.

Zemin Huang, Zhiyang Chen, Zijun Wang, Tiancheng Li, and Guo-Jun Qi. Reinforcing the diffusion chain of lateral thought with diffusion language models. In *Advances in Neural Information Processing Systems 39*, San Diego, CA, 2025.

Daniel Israel, Guy Van den Broeck, and Aditya Grover. Accelerating diffusion LLMs via adaptive parallel decoding. In *Advances in Neural Information Processing Systems 39*, San Diego, CA, 2025.

Yuchu Jiang, Yue Cai, Xiangzhong Luo, Jiale Fu, Jiarui Wang, Chonghan Liu, and Xu Yang. d^2Cache: Accelerating diffusion-based LLMs via dual adaptive caching. In *Proceedings of the 14th International Conference on Learning Representations*, Rio de Janeiro, Brazil, 2026.

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

Guanghao Li, Zhihui Fu, Min Fang, Qibin Zhao, Ming Tang, Chun Yuan, and Jun Wang. DiffuSpec: Unlocking diffusion language models for speculative decoding. *CoRR*, abs/2510.02358, 2025a.

Tianyi Li, Mingda Chen, Bowei Guo, and Zhiqiang Shen. A survey on diffusion language models. *CoRR*, abs/2508.10875, 2025b.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *Proceedings of the 12th International Conference on Learning Representations*, Vienna, Austria, 2024.

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. *CoRR*, abs/2506.06295, 2025.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 32819–32848, Vienna, Austria, 2024.

Guanxi Lu, Hao Mark Chen, Yuto Karashima, Zhican Wang, Daichi Fujiki, and Hongxiang Fan. AdaBlock-dLLM: Semantic-aware diffusion LLM inference via adaptive block size. In *Proceedings of the 14th International Conference on Learning Representations*, Rio de Janeiro, Brazil, 2026.

Lizhuo Luo, Shenggui Li, Yonggang Wen, and Tianwei Zhang. DSB: dynamic sliding block scheduling for diffusion LLMs. *CoRR*, abs/2602.05992, 2026a.

Lizhuo Luo, Zhuoran Shi, Jiajun Luo, Zhi Wang, Shen Ren, Wenya Wang, and Tianwei Zhang. DAWN: dependency-aware fast inference for diffusion LLMs. *CoRR*, abs/2602.06953, 2026b.

Xinyin Ma, Runpeng Yu, Gongfan Fang, and Xinchao Wang. dKV-Cache: The cache for diffusion language models. In *Advances in Neural Information Processing Systems 39*, San Diego, CA, 2025.

Quan Nguyen-Tri, Mukul Ranjan, and Zhiqiang Shen. Attention is all you need for KV cache in diffusion LLMs. In *Proceedings of the 14th International Conference on Learning Representations*, Rio de Janeiro, Brazil, 2026.

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

Yingte Shu, Yuchuan Tian, Chao Xu, Yunhe Wang, and Hanting Chen. Deferred commitment decoding for diffusion language models. *CoRR*, abs/2601.02076, 2026.

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 40th AAAI Conference on Artificial Intelligence*, pp. 33038–33046, Singapore, Singapore, 2026.

Lipeng Wan, Junjie Ma, Jianhui Gu, Zeyang Liu, Xuyang Lu, and Xuguang Lan. GeoBlock: Inferring block granularity from dependency geometry in diffusion language models. *CoRR*, abs/2603.26675, 2026.

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 *Proceedings of the 14th International Conference on Learning Representations*, Rio de Janeiro, Brazil, 2026.

Chengyue Wu, Hao Zhang, Shuchen Xue, Shizhe Diao, Yonggan Fu, Zhijian Liu, Pavlo O. Molchanov, Ping Luo, Song Han, and Enze Xie. Fast-dLLM v2: Efficient block-diffusion LLM. In *Proceedings of the 14th International Conference on Learning Representations*, Rio de Janeiro, Brazil, 2026a.

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 *Proceedings of the 14th International Conference on Learning Representations*, Rio de Janeiro, Brazil, 2026b.

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

Runpeng Yu, Qi Li, and Xinchao Wang. Discrete diffusion in large language and multimodal models: A survey. *CoRR*, abs/2506.13759, 2025.

Yu Zhang, Xinchen Li, Jialei Zhou, Hongnan Ma, Zhongwei Wan, Yiwei Shi, Duoqian Miao, Qi Zhang, and Longbing Cao. Swordsman: Entropy-driven adaptive block partition for efficient diffusion language models. *CoRR*, abs/2602.04399, 2026.

Siyan Zhao, Devaansh Gupta, Qinqing Zheng, and Aditya Grover. d1: Scaling reasoning in diffusion large language models via reinforcement learning. In *Advances in Neural Information Processing Systems 39*, San Diego, CA, 2025.

Fengqi Zhu, Rongzhen Wang, Shen Nie, Xiaolu Zhang, Chunwei Wu, Jun Hu, Jun Zhou, Jianfei Chen, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. LLaDA 1.5: Variance-reduced preference optimization for large language diffusion models. *CoRR*, abs/2505.19223, 2025a.

Fengqi Zhu, Zebin You, Yipeng Xing, Zenan Huang, Lin Liu, Yihong Zhuang, Guoshan Lu, Kangyu Wang, Xudong Wang, Lanning Wei, Hongrui Guo, Jiaqi Hu, Wentao Ye, Tieyuan Chen, Chenchen Li, Chengfu Tang, Haibo Feng, Jun Hu, Jun Zhou, Xiaolu Zhang, Zhenzhong Lan, Junbo Zhao, Da Zheng, Chongxuan Li, Jianguo Li, and Ji-Rong Wen. LLaDA-MoE: A sparse moe diffusion language model. *CoRR*, abs/2509.24389, 2025b.

A Dependency-Score Implementation

DepGA-Block compares model predictions before and after decoding the last block. These cross-step predictions approximate the idealized conditionals, which need not coincide with the model's conditionals under a single joint distribution. The implementation caches the top-K token probabilities with $K = 64$ and approximates KL using the current top-K tokens. For tokens absent from the previous cache, it uses a uniform probability over the remaining vocabulary based on the cached tail mass. This truncated calculation is a computational proxy for the full KL divergence.

The influence and entropy signals each undergo a centered three-position moving average with zero padding, followed by separate robust normalization within the candidate window:

$$\mathrm{Norm}(x_k) = \frac{x_k - \mathrm{median}(x)}{\max(\mathrm{IQR}(x), 10^{-15})}. \tag{9}$$

Here IQR is the interquartile range. DepGA-Block then computes $S_k = \tilde{I}_k - \lambda \tilde{H}_k$. Since normalized uncertainty can be negative, increasing λ does not necessarily shorten the selected block.

The first-negative-score rule provides a local boundary decision, with the length clipped to the allowed range. It is not equivalent to the cumulative criterion in Appendix B, since a negative per-position score can follow positive scores whose sum remains positive. The analysis motivates the score's structure, while the normalization and stopping rule make it practical for inference.

B Detailed Theoretical Analysis

B.1 Idealized Distributions and Expected Influence

Fix the decoded context c and a joint distribution $p(B, Z_{1:L} \mid c)$ over finite-valued sequences, where B is the preceding block's token content and $Z_{1:L}$ is the candidate next block. All marginal and conditional distributions in this analysis come from the same joint. For b with $p(b \mid c) > 0$, define

$$J_k(b) = \mathrm{KL}(p(Z_k \mid c, b) \,\|\, p(Z_k \mid c)), \qquad h_k(b) = \mathrm{H}(p(Z_k \mid c, b)). \tag{10}$$

The marginal is the mixture $p(z_k \mid c) = \sum_b p(b \mid c) p(z_k \mid c, b)$, so positive conditional probabilities have positive marginal support. Expanding the KL definition gives

$$\begin{aligned} \mathbb{E}_B[J_k(B)] &= \sum_{b, z_k} p(b, z_k \mid c) \log \frac{p(z_k \mid c, b)}{p(z_k \mid c)} \\ &= \mathrm{I}(Z_k; B \mid c) = I_k^{\mathrm{exp}}. \end{aligned} \tag{11}$$

Likewise, $\bar{h}_k = \mathbb{E}_B[h_k(B)] = \mathrm{H}(Z_k \mid c, B)$. Conditional entropy here already averages over B and requires no further expectation.

B.2 Exact Additivity Correction

Define the total correlations before and after conditioning on B by

$$\begin{aligned} T_0(L) &= \sum_{k=1}^{L} \mathrm{H}(Z_k \mid c) - \mathrm{H}(Z_{1:L} \mid c), \\ T_1(L) &= \sum_{k=1}^{L} \mathrm{H}(Z_k \mid c, B) - \mathrm{H}(Z_{1:L} \mid c, B). \end{aligned} \tag{12}$$

The conditional quantities in T_1 average over B. Both total correlations are nonnegative, since they equal the expected KL divergence between a joint distribution and the product of its marginals.

Proposition 1 (Additivity correction)**.** *For the joint distribution above,*

$$\mathcal{I}(L) = \mathrm{I}(Z_{1:L}; B \mid c) = \sum_{k=1}^{L} I_k^{\mathrm{exp}} - \varepsilon(L), \qquad \varepsilon(L) = T_0(L) - T_1(L). \tag{13}$$

Consequently, $|\varepsilon(L)| \le \max\{T_0(L), T_1(L)\}$.

Proof. Using $\mathrm{I}(U; B \mid c) = \mathrm{H}(U \mid c) - \mathrm{H}(U \mid c, B)$, subtract the joint influence from the sum of marginal influences:

$$\begin{aligned}
&\sum_{k=1}^{L} \mathrm{I}(Z_k; B \mid c) - \mathrm{I}(Z_{1:L}; B \mid c) \\
&= \left[\sum_{k=1}^{L} \mathrm{H}(Z_k \mid c) - \mathrm{H}(Z_{1:L} \mid c)\right] \\
&\quad - \left[\sum_{k=1}^{L} \mathrm{H}(Z_k \mid c, B) - \mathrm{H}(Z_{1:L} \mid c, B)\right] \\
&= T_0(L) - T_1(L).
\end{aligned} \tag{14}$$

Rearranging proves the identity. Nonnegativity of T_0 and T_1 gives the bound. □

Exact additivity holds when $T_0(L) = T_1(L)$, and approximate additivity holds when their difference is small. Independence of future positions both before and after conditioning on B is a sufficient, but stronger, condition. The correction is signed and accounts for both redundant and synergistic information. A short candidate window alone does not imply a small correction. The bound exposes the source of approximation error without asserting that it is uniformly small.

B.3 Connection to the Dependency Score

An idealized design criterion asks whether joint last-block influence dominates cumulative predictive uncertainty:

$$\mathcal{I}(L) \geq \lambda \sum_{k=1}^{L} \bar{h}_k. \tag{15}$$

Let $A(L) = \sum_{k=1}^{L} (I_k^{\mathrm{exp}} - \lambda \bar{h}_k)$. Equation (13) makes the exact criterion equivalent to $A(L) \geq \varepsilon(L)$, whereas dropping the correction gives $A(L) \geq 0$.

Proposition 2 (Sufficient condition for decision agreement)**.** *For a fixed candidate length L, if $|\varepsilon(L)| < |A(L)|$, the exact and additive criteria give the same decision.*

Proof. If $A(L) > 0$, then $\varepsilon(L) \leq |\varepsilon(L)| < A(L)$, so both criteria hold. If $A(L) < 0$, then $\varepsilon(L) \geq -|\varepsilon(L)| > A(L)$, so both fail. The strict premise excludes $A(L) = 0$. □

This result explains when the additive approximation preserves the idealized cumulative decision. Its per-position terms motivate the influence-minus-uncertainty dependency score.

C Evaluation Protocol

All benchmark evaluations use `lm-eval` 0.4.11 with the shot settings in Section 5. Main-table results use one complete evaluation pass with temperature zero and batch size 1. Generation uses greedy token predictions without sampling noise. Math-500 uses the `minerva_math500` task, with the additional long-generation evaluation reporting its `math_verify` accuracy. HumanEval accuracy is pass@1 after Python-code extraction, sanitization, and execution of the benchmark tests. Dream HumanEval responses are preserved until code extraction and sanitization.

Throughput measurement. TPS is the total number of generated tokens divided by the elapsed time of the complete generation loop, including input preparation, model execution, decoding-policy computation, and output processing. Timing starts after model initialization and request batching and ends after final output handling. Model loading and downstream task scoring are excluded. NFE is the mean number of model forwards per example.

Shared configuration. We use LLaDA-8B-Instruct/GSM8K for one-time calibration and select $\tau_{\text{low}} = 0.8$ as an intermediate accuracy-throughput setting, then keep it fixed for the other 15 backbone-task combinations. The same default configuration applies across the four backbones and four tasks, spanning the LLaDA and Dream families, dense and MoE architectures, and mathematical reasoning and code generation. The hyperparameter-analysis sweeps characterize decoding behavior under this shared configuration.

D Experimental Results on an MoE Backbone

To further examine whether DepCap generalizes beyond dense DLM backbones, we conduct an additional set of experiments on LLaDA-MoE-7B-A1B-Instruct, an instruction-tuned diffusion language model with a mixture-of-experts architecture. The evaluation follows the same setting as Section 5. Unless otherwise specified, the hyperparameters of Fast-dLLM, AdaBlock, Swordsman, and DepCap are kept identical to those used in the main experiments. For Fast-dLLM, we use a fixed initial block size of $|B_0| = 32$.

Table 2 reports the complete results. DepCap achieves the lowest NFE on GSM8K, Math-500, and HumanEval. The quality-speed trade-off is particularly favorable on HumanEval, where DepCap combines the highest accuracy with the second-highest throughput. On the mathematical reasoning tasks, Fast-dLLM retains higher accuracy, while DepCap uses fewer model evaluations. On MBPP, DepCap is close to the baselines in throughput but does not improve accuracy. These results extend the evaluation to an MoE backbone and show a task-dependent trade-off.

Table 2: Additional results on LLaDA-MoE-7B-A1B-Instruct across four benchmarks. We report task accuracy (Acc), throughput in tokens per second (TPS), and number of function evaluations (NFE). Best results are **bolded**, and second-best are underlined. Ties use the reported precision.

LLaDA-MoE-7B-A1B-Instruct												
Method	**GSM8K (5-shot)**			**Math-500 (4-shot)**			**MBPP (3-shot)**			**HumanEval (0-shot)**		
	TPS ⇑	NFE ⇓	Acc ⇑	TPS ⇑	NFE ⇓	Acc ⇑	TPS ⇑	NFE ⇓	Acc ⇑	TPS ⇑	NFE ⇓	Acc ⇑
Fast-dLLM	**11.8**	93.5	**77.4**	**10.5**	104.8	**41.2**	**12.4**	61.9	54.0	**18.4**	61.7	51.8
Swordsman	8.7	122.9	64.9	8.8	116.0	30.6	12.2	**58.4**	46.8	16.4	66.1	53.7
AdaBlock	10.1	100.0	76.4	9.3	113.6	39.6	10.8	67.1	**54.4**	15.7	71.2	56.7
DepCap	**11.8**	**89.7**	76.7	10.3	**101.4**	40.4	12.1	64.2	52.6	17.6	**61.4**	**57.3**

E Case Study

To complement the aggregate experimental results, Figure 3 shows two GSM8K examples that illustrate the two inference decisions targeted by DepCap: how to determine block boundaries and how to parallelize decoding within each block. Both methods use the same prompt, backbone, and generation length to compare their decoding trajectories under a shared setup. DepCap adapts its block sizes across different stages of generation. By combining last-block influence with predictive uncertainty, DepGA-Block adapts the block boundaries to changes in the model's predictions across decoding steps. Inside each selected block, CAP-Decoding uses pairwise conflict scores to filter candidates for parallel decoding.

Fast-dLLM instead uses a fixed 32-token block schedule. The red spans show reasoning errors in the two examples. In Case A, Fast-dLLM mixes the total value of all lego sets with the revenue needed for the purchase. In Case B, it compares the boosted range with the original range instead of the dragon's flame range. DepCap produces the correct final answers in these examples. These cases complement the quantitative results with examples of block layouts and decoding trajectories.

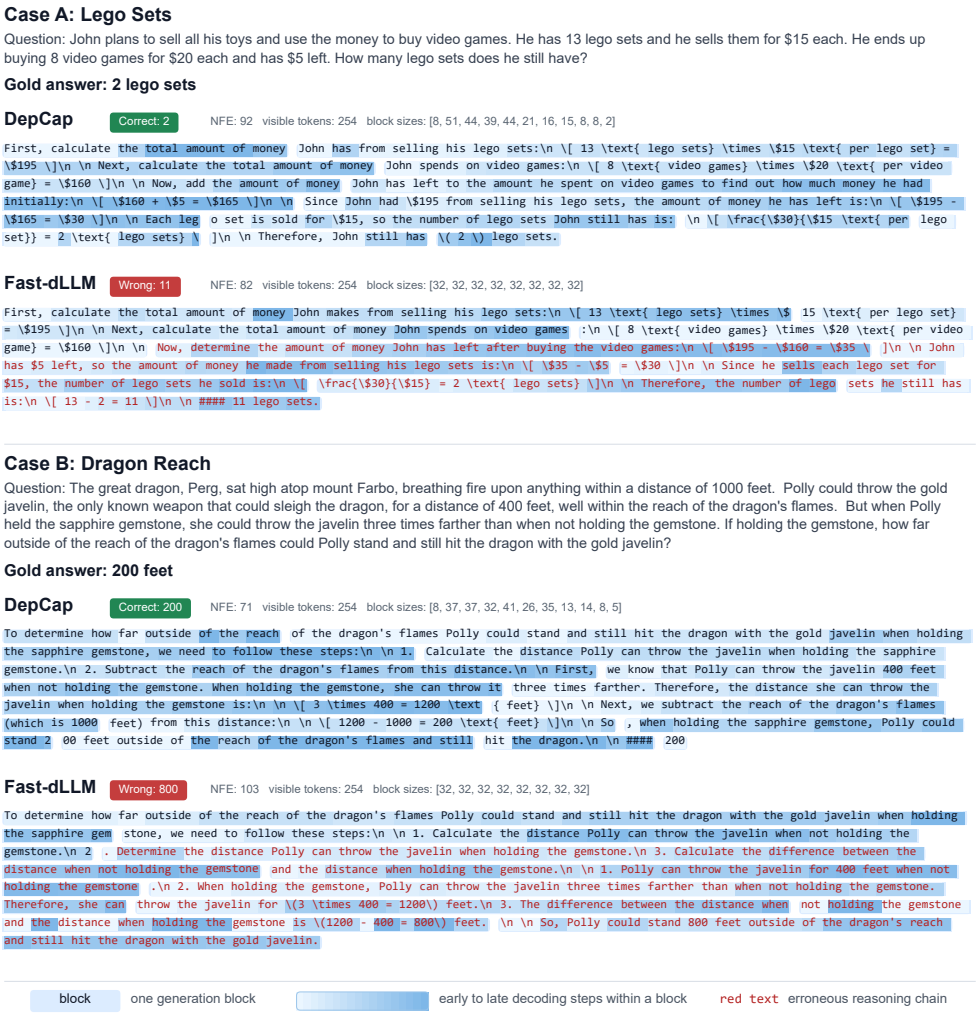


Figure 3: GSM8K case studies on LLaDA-8B-Instruct. Light-blue spans denote generation blocks, with gaps separating consecutive blocks. Within each block, lighter-to-darker blue indicates earlier-to-later decoding steps. Red text marks erroneous reasoning chains in the Fast-dLLM output.

F Runtime Profile and Repeatability

F.1 Component-Level Profile

We profile DepCap on LLaDA-8B-Instruct using a fixed 128-example GSM8K subset, batch size 1, and the default generation configuration. Table 3 averages CUDA-event timings over 9,503 decoding iterations. Model forward accounts for most of the measured GPU time. The explicitly instrumented DepCap-specific operations other than model forward and confidence scoring total 0.883 ms per iteration, or 0.568% of the measured time (aggregated before rounding). GPU-time shares differ from end-to-end overhead, which TPS already includes.

In the profiled DepCap run, peak allocated CUDA memory is 18.907 GiB, with a maximum generation-time increment of 3.965 GiB relative to the post-model-loading allocation. Peak reserved memory is 37.756 GiB. Including lookahead, the top-K cache uses $O(L_{\max}K)$ storage.

Table 3: Component-level CUDA-event profile of DepCap at batch size 1. Shares use the total measured GPU time, including unattributed event gaps.

Component	ms/iteration	GPU-time share (%)
Model forward	147.884	95.193
Confidence scoring	6.003	3.864
Top-K cache construction	0.316	0.203
Approximate KL	0.026	0.017
Entropy	0.146	0.094
Robust normalization	0.033	0.021
Conflict matrix	0.102	0.066
Subset selection	0.187	0.121
Force-progress handling	0.072	0.047
Unattributed event gaps	0.582	0.374

F.2 Candidate-Pool Scaling

The observed candidate-pool size has mean 3.94, median 2, 95th percentile 14, and maximum 67. Table 4 additionally measures the CAP-Decoding routine for controlled pool sizes, using 20 repetitions at batch size 1. Larger sparse-conflict pools require more greedy selections, whereas dense conflicts remove more candidates per selection and terminate the loop earlier. This explains why runtime depends on the conflict pattern as well as the number of candidates. The conflict matrix requires $O(n^2)$ storage for a pool of size n.

Table 4: Candidate-pool microbenchmark at batch size 1. Times are mean milliseconds per CAP-Decoding call across 20 repetitions, including confidence scoring, conflict scoring, subset selection, and force-progress handling.

Pool size	Sparse-conflict time	Dense-conflict time
16	2.832	1.083
32	4.698	1.080
64	8.284	1.058
128	15.753	1.413

F.3 Process-Level Repeatability

We repeat the default DepCap evaluation in five fresh processes on the same GSM8K subset. The runs fix Python, NumPy, and PyTorch seeds to 1234, enable `cudnn.deterministic`, disable `cudnn.benchmark`, and use evaluation mode with temperature zero. All five runs produce identical generated token sequences, 29,450 generated tokens, and 9,503 total model evaluations. Accuracy and NFE are therefore unchanged in this check.

Table 5 reports the timing variation across the fixed-workload runs. Mean TPS is 20.1103 with sample standard deviation 0.1714 and coefficient of variation 0.852%. The 95% Student-t interval for mean TPS is $[19.8974, 20.3231]$.

Table 5: Five fresh-process runs of DepCap on the same GSM8K subset. Each run uses the default configuration on LLaDA-8B-Instruct.

Run	TPS	Average NFE
1	20.4147	74.2422
2	19.9990	74.2422
3	20.0510	74.2422
4	20.0435	74.2422
5	20.0432	74.2422

G Cross-Setting Component Comparisons

The component rows in Table 1 distinguish the effects of block partitioning and parallel decoding. Relative to confidence decoding with a fixed block size of 32, adding DepGA-Block raises accuracy in seven settings, matches it in one, and lowers it in four. Replacing confidence decoding with fixed-block CAP-Decoding increases TPS in 11 of 12 settings. Relative to fixed-block CAP-Decoding, the full method raises accuracy in nine settings, matches it in one, and lowers it in two. These comparisons show the components' different effects on accuracy and throughput across settings.

H Hyperparameter and Long-Generation Analysis

We analyze five parameters on LLaDA-8B-Instruct/GSM8K, extend the τ_{low} and λ sweeps across backbones and tasks, and compare four methods at $L_{\text{gen}} = 1024$. Each sweep varies one parameter with the others fixed. Acc is percentage accuracy, TPS measures generated tokens per second, and NFE averages model evaluations per example.

H.1 Generation Length

Table 6: Effect of generation length L_{gen} on DepCap on LLaDA-8B-Instruct over GSM8K.

L_{gen}	Acc ⇑	TPS ⇑	NFE ⇓
128	74.5	19.1	48.4
256	78.8	21.4	74.8
512	79.8	15.5	99.2
1024	77.0	8.7	120.4

Table 6 shows that longer generations require more model evaluations. Throughput peaks at 256 tokens before declining, while accuracy improves up to 512 tokens and then decreases at 1024. Thus, a larger generation budget does not necessarily improve task performance. Appendix H.7 compares four decoding methods at 1024 tokens on Math-500 and HumanEval.

H.2 Maximum Block Size

Table 7: Effect of the maximum block size $L_{\max}$ on DepCap on LLaDA-8B-Instruct over GSM8K.

$L_{\max}$	Acc ⇑	TPS ⇑	NFE ⇓
16	79.6	17.6	90.8
32	80.1	19.1	83.6
64	78.5	20.7	77.5
128	78.8	21.4	74.8
256	79.5	21.5	74.8

Table 7 shows that increasing $L_{\max}$ improves TPS and reduces NFE until the efficiency gains saturate near 128. Small values restrict the available parallelism, whereas increasing the cap further has little effect on NFE. Accuracy varies non-monotonically, indicating that the cap primarily determines the available parallelism rather than directly controlling generation quality.

H.3 Uncertainty Weight

The uncertainty weight λ balances locally normalized last-block influence and predictive uncertainty. Table 8 shows relatively small variation in accuracy, TPS, and NFE across the tested range. Larger λ does not produce a monotonic change in either quality or efficiency, consistent with its effect on the normalized score rather than a direct constraint on block length.

Table 8: Effect of the uncertainty weight λ on DepCap on LLaDA-8B-Instruct over GSM8K.

λ	Acc ⇑	TPS ⇑	NFE ⇓
0.6	78.8	20.6	75.1
0.8	78.9	20.7	74.9
1.0	79.0	20.8	74.8
1.2	78.8	21.4	74.8
1.4	78.8	20.7	74.8
1.6	78.6	20.8	74.9

Table 9: Effect of the confidence threshold τ_{low} on DepCap on LLaDA-8B-Instruct over GSM8K.

τ_{low}	Acc ⇑	TPS ⇑	NFE ⇓
0.65	78.4	24.8	61.7
0.70	78.5	24.3	65.5
0.75	78.7	18.4	69.8
0.80	78.8	21.4	74.8
0.85	80.3	19.0	80.3

H.4 Confidence Threshold

The confidence threshold τ_{low} controls the candidate pool before conflict filtering. In Table 9, increasing the threshold improves accuracy and increases NFE, reflecting more selective parallel decoding. TPS is lower at the high-threshold endpoint but does not vary monotonically at every intermediate value. Appendix H.6 examines this behavior across backbones and tasks.

H.5 Conflict Threshold

Table 10: Effect of the conflict threshold γ on DepCap on LLaDA-8B-Instruct over GSM8K.

γ	Acc ⇑	TPS ⇑	NFE ⇓
-8.0	78.9	23.8	67.2
-12.0	78.4	22.8	70.1
-16.0	78.8	21.4	74.8
-20.0	79.5	19.9	80.6
-24.0	79.5	17.8	86.3

As shown in Table 10, the conflict threshold γ determines how strictly CAP-Decoding removes conflicting candidates inside the current block. A looser threshold improves throughput, with the best TPS achieved at $\gamma = -8.0$, while more conservative thresholds gradually reduce speed and increase NFE. Accuracy remains relatively stable, suggesting that γ mainly controls within-block parallelism after candidate-pool construction.

H.6 Cross-Model and Cross-Task Hyperparameter Analysis

We vary τ_{low} and λ separately on LLaDA-8B-Instruct and Dream-v0-base-7B, using complete Math-500 and HumanEval evaluations at $L_{\text{gen}} = 256$. All other settings remain at their defaults. Tables 11 and 12 report every tested configuration, with the shared default included in both sweeps.

Confidence threshold. On LLaDA, raising τ_{low} increases NFE and lowers TPS on both tasks. HumanEval accuracy improves, whereas Math-500 accuracy varies non-monotonically. On Dream, accuracy and average NFE remain unchanged over the same threshold range on both tasks, with small throughput fluctuations. Thus, the threshold affects decoding differently across backbones.

Table 11: Confidence-threshold sweep at $L_{\text{gen}} = 256$. Each panel reports complete Math-500 and HumanEval results. The shared default is $\tau_{\text{low}} = 0.80$. Acc is in percent.

	Math-500			HumanEval		
τ_{low}	Acc	TPS	NFE	Acc	TPS	NFE
LLaDA-8B-Instruct						
0.65	38.20	25.654	79.442	39.63	67.644	63.878
0.70	37.60	24.273	83.754	41.46	64.238	66.780
0.75	40.20	22.978	89.190	42.68	61.576	69.988
0.80	39.80	21.574	94.796	42.68	57.147	74.787
0.85	39.80	20.228	101.218	43.90	53.190	80.226
Dream-v0-base-7B						
0.65	40.80	24.330	97.150	50.61	40.353	125.293
0.70	40.80	24.332	97.150	50.61	41.387	125.293
0.75	40.80	24.365	97.150	50.61	40.709	125.293
0.80	40.80	24.364	97.150	50.61	40.364	125.293
0.85	40.80	24.339	97.150	50.61	41.346	125.293

Uncertainty weight. Varying λ produces smaller changes in decoding efficiency than the LLaDA confidence-threshold sweep. The largest within-setting spans are 1.494 TPS and 1.524 average NFE across the tested values, while accuracy has no consistent monotonic trend. On Dream HumanEval, the shared default attains 50.61% accuracy, and the other tested values reach 51.22–51.83%.

Table 12: Uncertainty-weight sweep at $L_{\text{gen}} = 256$. Each panel reports complete Math-500 and HumanEval results. The shared default is $\lambda = 1.2$. Acc is in percent.

	Math-500			HumanEval		
λ	Acc	TPS	NFE	Acc	TPS	NFE
LLaDA-8B-Instruct						
0.6	39.20	21.802	93.968	41.46	56.047	76.311
0.8	40.00	21.710	94.214	42.07	56.828	75.146
1.0	39.80	21.573	95.018	43.90	57.160	75.024
1.2	39.80	21.574	94.796	42.68	57.147	74.787
1.4	41.20	21.297	94.824	42.68	57.541	75.280
1.6	40.60	21.315	94.784	42.07	57.307	75.348
Dream-v0-base-7B						
0.6	40.60	24.142	98.050	51.22	40.875	124.829
0.8	40.80	24.169	97.864	51.22	40.536	125.116
1.0	40.60	24.274	97.434	51.22	41.293	125.189
1.2	40.80	24.364	97.150	50.61	40.364	125.293
1.4	40.60	24.278	97.534	51.22	40.640	124.598
1.6	40.40	24.210	97.594	51.83	40.763	125.226

H.7 Long-Generation Comparisons

Table 13 compares four methods at $L_{\text{gen}} = 1024$ on LLaDA-8B-Instruct. The evaluation covers all 500 Math-500 examples with 4-shot prompting and all 164 HumanEval examples with 0-shot prompting. Fast-dLLM uses a fixed block size of 32, and the other method settings follow Section 5.

DepCap maintains a competitive trade-off at the longer generation length. On Math-500, its accuracy lies between Fast-dLLM and AdaBlock, while its throughput exceeds AdaBlock's. On HumanEval, DepCap achieves the highest throughput and matches AdaBlock's accuracy with fewer model evaluations. Fast-dLLM is slightly more accurate, whereas Swordsman's lower NFE does not translate into higher throughput. These comparisons show why NFE and measured throughput provide complementary efficiency information.

Table 13: Long-generation results on LLaDA-8B-Instruct with $L_{\text{gen}} = 1024$. Acc is percentage accuracy, TPS counts generated tokens only, and NFE is averaged over the complete benchmark. HumanEval accuracy uses the code-postprocessing protocol in Appendix C.

Method	Acc (%) ↑	TPS ↑	NFE ↓
Math-500 (4-shot)			
DepCap	42.20	13.894	201.712
Fast-dLLM	41.00	14.678	193.308
AdaBlock	44.40	13.286	210.114
Swordsman	34.20	12.211	194.040
HumanEval (0-shot)			
DepCap	44.51	19.174	192.067
Fast-dLLM	45.12	18.543	197.628
AdaBlock	44.51	17.034	214.604
Swordsman	23.78	11.495	167.793

AdaBlock's HumanEval TPS excludes prompt tokens.