ASYNCLANE: DECOUPLING REFINEMENT FROM ADVANCEMENT IN DIFFUSION LANGUAGE MODEL DECODING

Anonymous authors
Paper under double-blind review

ABSTRACT

Block decoding in diffusion language models starts the next block only after the current block is fully completed, letting a few unresolved positions delay all subsequent advancement. In the serial decoding trajectories we analyze, some next-bl ock predictions already match those obtained after current-block completion. Our key insight is that completion requires resolving every pending position whereas advancement can begin from a subset of candidates, so the two can be decoupled. We introduce AsyncLane, a training-free scheduler in which the main lane completes the current block while a private lane accumulates next-block progress in the same batched forward pass. Private progress is handed off after completion; retained candidates condition subsequent private-lane predictions, and lookahead prefill (LA) reuses full-sequence predictions from the opening step. We evaluate AsyncLane on LLaDA, Dream, and DiffusionGemma across five benchmarks. Under the reported configurations, effective output throughput on LLaDA and Dream reaches up to 2.35× that of AdaBlock, and on Diffusion Gemma up to 1.23× its baseline. Quality changes vary by task, with gains in both quality and throughput on multiple tasks; for example, Dream's MATH500 accuracy exceeds AdaBlock's by 3.80 percentage points. Controlled ablations on LLaDA and Dream show that early opening reduces request latency and that full LA provides an additional reduction under fixed early opening.

1 INTRODUCTION

Diffusion language models generate text through iterative denoising, updating multiple token positions in parallel (Nie et al., 2025; Ye et al., 2025). Efficient implementations organize these updates into blocks and reuse approximate caches (Wu et al., 2025). Yet each block starts only after every MASK in the previous block is filled. This block-completion barrier lets a few unresolved positions delay all subsequent decoding, raising a scheduling question: *must every position in the next block wait for the current block to finish?*

Figure 1(a) compares next-block predictions before and after filling only the remaining MASKs in the current block: some agree, while others change. Independently of our scheduler, we quantify this behavior on standard serial decoding trajectories, observing next-block predictions without writing them back and comparing them with predictions after current-block completion. Figure 2 groups counts of matching positions by the forward calls still required to complete the current block. Across 200 prompts from four benchmarks on LLaDA-8B-Instruct, 71.5% of observed states have at least one matching non-special next-block prediction while the current block still requires at least two forward calls. This retrospective agreement does not identify which predictions can be accepted online, but motivates selective advancement before completion.

Completing the current block requires resolving every remaining position; next-block progress can begin with a subset. We therefore *decouple current-block completion from next-block advancement*, allowing both activities to overlap.

Existing approaches organize progress across blocks differently. D2F introduces new blocks according to the preceding block's completion fraction on a distilled block-causal model (Wang et al., 2025), while AHD uses historical consistency of prediction distributions to select cross-block po-

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

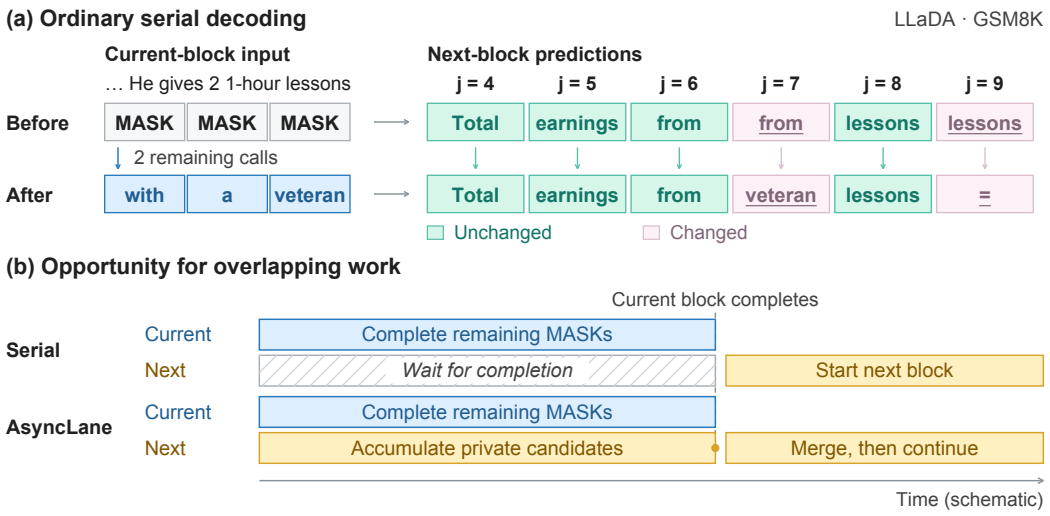


Figure 1: **Prediction agreement and overlapping progress.** (a) Four of six displayed next-block predictions match across current-block completion (LLaDA, GSM8K). (b) Private candidates accumulate before completion and merge afterward (schematic). Agreement does not imply correctness.

sitions for direct updates to the generation sequence (Zou et al., 2026). We accumulate next-block candidates in a separate state, where they support further prediction without entering the input used to complete the current block.

AsyncLane implements this design as a training-free scheduler. At a chosen fill ratio, a full-sequence opening forward continues the current block and initializes next-block candidates. During overlap, the main lane completes the current block while the private lane selects candidates by confidence and uses them to condition further next-block predictions. Unresolved current-block positions remain MASK in the private view. The two views share a batched forward call, but candidates enter the main state only after current-block completion, without overwriting filled positions. AsyncLane also reuses eligible high-confidence opening-forward outputs for lookahead prefill (LA).

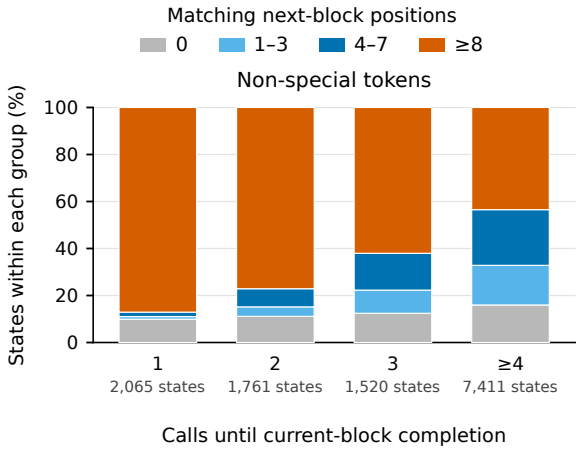


Figure 2: **Next-block prediction agreement.** Colors indicate counts of matching non-special predictions before the first output EOS. Bars group states by remaining calls, including the observation call.

We evaluate AsyncLane on LLaDA, Dream, and DiffusionGemma using five benchmarks covering mathematical reasoning, code generation, and instruction following. On LLaDA and Dream, AsyncLane achieves up to 2.35× the effective output throughput of AdaBlock under the reported configurations. Higher throughput is accompanied by higher quality on multiple tasks; for example, Dream's MATH500 accuracy exceeds AdaBlock's by 3.80 percentage points. On DiffusionGemma, AsyncLane achieves up to 1.23× the effective output throughput of the official decoder; quality changes remain task-dependent across all three models. To isolate the contribution of the scheduling design, we conduct controlled ablations on LLaDA and Dream. Early opening reduces request latency, supporting the practical value of overlapping current-block completion with next-block advancement; with early opening held fixed, full LA provides a further latency reduction.

Our contributions are as follows:

- **Serial decoding analysis.** We quantify how many next-block predictions agree before and after current-block completion, and how much completion work remains when such agreement occurs.
- **AsyncLane.** We introduce a training-free scheduler that accumulates candidates in a private state, uses them for further prediction, and merges them after current-block completion.
- **Empirical results.** Across three models and five benchmarks, we report higher effective output throughput than Fast-dLLM and AdaBlock on LLaDA and Dream, and than the official decoder on DiffusionGemma, with higher quality on multiple tasks. Controlled ablations on LLaDA and Dream identify latency benefits from early opening and full LA, with task-dependent quality changes.

2 Related Work

Prior work accelerates diffusion language models through cache reuse and adaptive parallel acceptance (Wu et al., 2025; Liu et al., 2026; Israel et al., 2025; Bao et al., 2025). Block Diffusion combines autoregressive generation across blocks with diffusion within blocks, while AdaBlock-dLLM selects boundaries using predicted semantic delimiters and their confidence (Arriola et al., 2025; Lu et al., 2025).

Cross-block advancement. D2F adapts models to block-causal attention through distillation and pipelines blocks according to preceding-block progress (Wang et al., 2025). FlowBlock overlaps blocks on pretrained self-correcting models using block-causal attention and ordered commitment (Tian et al., 2026). AHD uses historical consistency and confidence to accept cross-block predictions directly into the generation state (Zou et al., 2026). Trajectory-Level Speculative Decoding constructs and verifies attention-isolated denoising trajectories and supports speculation across adjacent blocks (Pan et al., 2026). AsyncLane overlaps current-block refinement with next-block advancement without additional training, retaining bidirectional attention in LLaDA and Dream. Its candidates condition subsequent private predictions but remain absent from current-block inputs until completion, when they merge into the main state without a separate verification call. LA separately updates the main state directly.

3 Method

AsyncLane overlaps current-block refinement with private advancement of the next block. A full-sequence opening forward initiates this schedule and supplies predictions for lookahead prefill; private candidates are merged only after current-block completion. Figure 3 illustrates the cycle, Algorithm 1 summarizes the decoding loop, and Appendix C gives the complete algorithm.

3.1 From the Completion Barrier to Two Lanes

State and blocks. The decoder generates on a length-L canvas initialized to `MASK`, with the prompt as a fixed prefix. Its current content is the main state x, indexed by canvas position. Decoding proceeds left to right through the current block $A = [a, b)$ and next block $B = [c, d)$. Refinement fills pending positions of A; advancement retains partial predictions in B to condition subsequent predictions before writing them to the main state.

Block completion barrier. Let $m(A, x)$ be the number of remaining `MASK` positions in A; A is complete when $m(A, x) = 0$. The semi-autoregressive decoding of LLaDA and block decoders such as Fast-dLLM and AdaBlock-dLLM begin advancing B only after A is complete (Nie et al., 2025; Wu et al., 2025; Lu et al., 2025); we call this the block completion barrier.

Two lanes. AsyncLane decouples refinement from advancement by maintaining two lanes. The main lane refines A from x using Fast-dLLM's rule (Wu et al., 2025): positions whose predicted

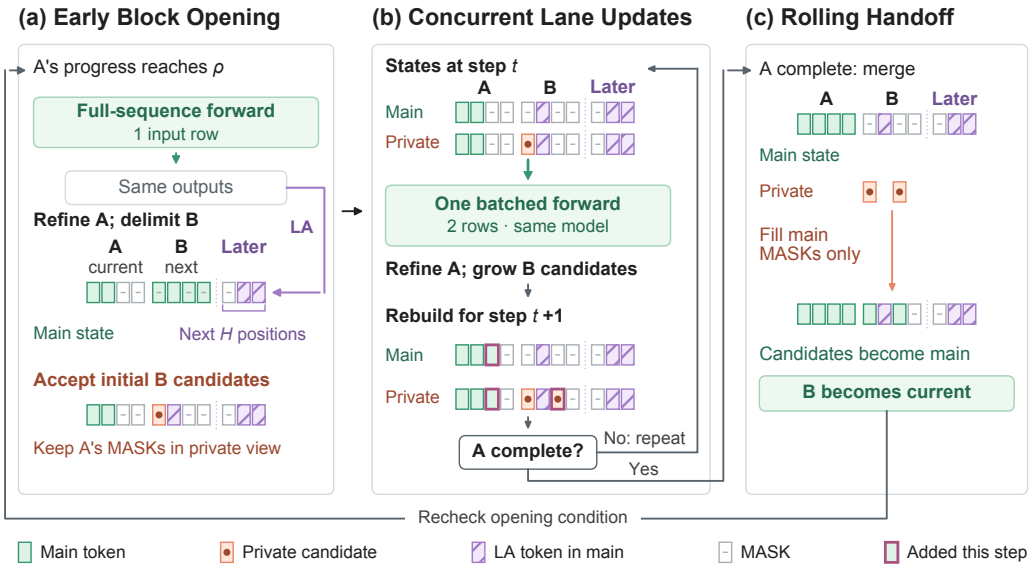


Figure 3: **AsyncLane decoding cycle: early opening, concurrent updates, and handoff.** A and B denote the current and next blocks. Rows depict logical states.

probability (confidence) reaches τ_{main} are committed; if none qualifies, the highest-confidence position is committed, ensuring progress. The private lane advances B before A completes (Section 3.2). Its private view fills accepted candidates into B while leaving pending positions of A as `MASK`. These predictions are screened and retained as candidates outside the main state (Section 3.3). Only after A completes are candidates merged into still-masked positions of B, which becomes the current block. This delay prevents candidates from conditioning unfinished positions of A under bidirectional attention; once merged, they are not revoked.

3.2 Early Opening

Block extent. A full-sequence pass, the opening forward, opens each block using AdaBlock-dLLM's delimiter-confidence rule (Lu et al., 2025). Block B starts at the first `MASK` position c past A, skipping consecutive lookahead-prefilled positions, which remain unchanged thereafter (Section 3.4). The opening forward searches a bounded window after c for predicted delimiters (e.g., newlines). If the highest-confidence delimiter reaches τ_{bd}, B ends at and includes it; otherwise its length is ℓ_0, capped at the canvas end. The configuration table gives each model's window and delimiter set. The private lane advances within B until handoff.

Opening timing. When no private lane is active, the decoder checks the opening condition before the next refinement step. The fill ratio of A is

$$g(A, x) = 1 - \frac{m(A, x)}{|A|}, \tag{1}$$

counting both prefilled positions and merged candidates. If $g(A, x) \geq \rho$ and `MASK` positions remain after A, the decoder opens B; $\rho = 1$ waits for completion. With early opening, this forward also refines A and supplies initial private candidates for B (Section 3.3), adding no extra model call.

3.3 Private Advancement and Post-Completion Merge

Constructing the private view. The private lane must predict the remaining positions of B conditioned on accepted candidates, without writing them to the main state before A completes. It therefore constructs a separate input, the private view. Let C record accepted private candidates in

Algorithm 1 AsyncLane decoding (simplified). T records revisable prefill. Cache management is omitted; see Appendix C for the complete algorithm.

Require: Model, prompt, canvas length L; thresholds $\tau_{\text{main}}, \rho, \tau_{\text{priv}}, \tau_{\text{floor}}, \tau_{\text{bd}}, \tau_{\text{LA}}$; fallback block length ℓ_0; horizon H

$x_i \leftarrow$ `MASK` for $i = 0, \ldots, L-1$; $P \leftarrow \text{OpeningForward}(x)$
$a \leftarrow 0$; $b \leftarrow \text{Boundary}(a, P)$; $A \leftarrow [a, b)$; $x \leftarrow \text{MainUpdate}(x, A, P)$
$T \leftarrow \varnothing$; $(x, T) \leftarrow \text{Lookahead}(x, T, P, b)$ {Content prefill and EOS padding}
loop
  $g \leftarrow 1 - m(A, x)/|A|$ {Eq. 1}
  if $g \geq \rho$ **and** `MASK` remains after A **then**
    $P \leftarrow \text{OpeningForward}(x)$; $x \leftarrow \text{MainUpdate}(x, A, P)$
    $c \leftarrow$ first `MASK` position $\geq b$; $d \leftarrow \text{Boundary}(c, P)$; $B \leftarrow [c, d)$
    $(x, T) \leftarrow \text{Lookahead}(x, T, P, d)$ {Reuse opening predictions}
    $C \leftarrow \text{Admit}(\varnothing, x, B, P)$ {Initial candidates}
    while $m(A, x) > 0$ **do**
      $\widetilde{x} \leftarrow$ private view of x and C {Eq. 2}
      $(P^{\text{main}}, P^{\text{priv}}) \leftarrow \text{OverlapCall}(x, \widetilde{x})$
      $x \leftarrow \text{MainUpdate}(x, A, P^{\text{main}})$; $C \leftarrow \text{Admit}(C, x, B, P^{\text{priv}})$
    end while
    $x_i \leftarrow C_i$ for $i \in \text{dom}(C)$ with $x_i =$ `MASK` {Handoff}
    $(a, b) \leftarrow (c, d)$; $A \leftarrow B$
  else if $m(A, x) > 0$ **then**
    $P \leftarrow \text{RefinementCall}(x)$; $x \leftarrow \text{MainUpdate}(x, A, P)$
  else
    break
  end if
end loop
return the text of x up to its first EOS

B, with token C_i at position i. At the start of each overlap call t, this view is built from the latest main state:

$$\widetilde{x}_i^t = \begin{cases} C_i, & i \in \text{dom}(C) \text{ and } x_i^t = \texttt{MASK}, \\ x_i^t, & \text{otherwise.} \end{cases} \tag{2}$$

Here $\text{dom}(C)$ contains positions with accepted candidates. Pending positions of A remain `MASK` in both the main state and the private view, without temporary predictions. Lookahead prefill in the main state appears in both lanes' inputs; unmerged candidates appear only in the private view.

Update and candidate acceptance. The main state and private view are computed as two batch rows in a single model call. The main lane refines A by the rule of Section 3.1. The private lane considers only the following positions:

$$U_B = \{i \in B : x_i = \texttt{MASK},\ i \notin \text{dom}(C)\}, \tag{3}$$

i.e., still-masked positions of B without an accepted candidate. For each $i \in U_B$, let $\hat{y}_i$ be the private lane's highest-probability token and q_i its probability. Positions with $q_i \geq \tau_{\text{priv}}$ are added to C. If none qualifies and U_B is nonempty, only the highest-confidence position is accepted, and only if its probability reaches τ_{floor}; otherwise no candidate is added. Existing candidates are retained and condition subsequent private predictions. Initial candidates from the opening forward use the same acceptance rule. Because private candidates are formed while A is still incomplete, they use a separate threshold τ_{priv}. Refinement progress is guaranteed by the main lane, so the private lane need not add a candidate every round; hence the fallback accepts one only when the highest confidence is at least τ_{floor}.

Post-completion handoff and rolling. When $m(A, x) = 0$, candidates in C fill still-masked positions in the main state without overwriting existing content. If the opening forward completes A, initial candidates merge immediately, with no overlap calls. Block B becomes current, the candidate record is cleared, and the cycle repeats. Handoff requires no additional verification forward.

Prefill and merged candidates both count toward B's fill ratio; reaching ρ allows immediate opening of the next block. If no `MASK` remains after the current block, the decoder finishes refining it and stops, returning the text up to the first EOS.

3.4 Full-Sequence Prediction Reuse

Content prefill. Every opening forward, including the first, predicts beyond the opened blocks. Lookahead prefill (LA) reuses these outputs without a dedicated call. For each position i, let $\hat{y}_i$ be the highest-probability token and q_i its probability. The rightmost opened boundary is $r = b$ when only A is open and $r = d$ after opening B. Non-EOS content predictions enter the main state only when

$$x_i = \texttt{MASK}, \qquad r \le i < \min(r + H, L), \qquad q_i \ge \tau_{\mathrm{LA}}. \tag{4}$$

Unlike private candidates, prefill directly enters the shared main state and conditions subsequent computation. It therefore requires high confidence and a finite horizon H, with later rescoring and possible revocation.

Rescoring and revocation. When content prefill is written, its position is recorded only if the canvas contains no EOS. At each subsequent opening forward, the decoder determines the new block and updates r to its right boundary, then drops records left of r, including positions inside the new block or skipped by its start c. These tokens remain unchanged thereafter. For the remaining records, it recomputes the probability assigned to each recorded token and restores positions with probability below τ_{LA} to `MASK`. New prefill from the current forward is written after processing these records.

EOS padding. After the first EOS appears in the canvas, the opening forward also writes EOS at positions beyond the opened blocks that are predicted as EOS with confidence at least τ_{main}. These writes are not limited by H and are not revoked. This can reduce the number of subsequent decoding calls needed to fill the remaining canvas. The remaining EOS processing rules are in Appendix A.

3.5 Execution

The two lanes are two input rows in the same model call, sharing model weights. The cache follows the approximate KV cache of Fast-dLLM (Wu et al., 2025): the opening forward is a full-sequence call that refreshes the entire cache; subsequent calls recompute only the active interval, reusing cached keys and values at positions outside it. Each lane maintains its own cache copy. Because representations outside the interval also change with context under bidirectional attention, this reuse is approximate. The exact range of the active interval, cache handling at handoff, and model-specific implementation details are in Appendix B.

4 Experiments

4.1 Experimental Setup

Models and benchmarks. We evaluate three frozen diffusion language models: LLaDA-8B-Instruct (Nie et al., 2025), Dream-v0-Instruct-7B (Ye et al., 2025), and DiffusionGemma-26B-A4B-it (DiffusionGemma Team et al., 2026). LLaDA and Dream retain bidirectional attention; DiffusionGemma uses an encoder–decoder architecture with thinking enabled (adaptation in Appendix D). Benchmarks cover math reasoning (GSM8K (Cobbe et al., 2021), MATH500 (Hendrycks et al., 2021; Lightman et al., 2023)), code generation (HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021)), and instruction following (IFEval (Zhou et al., 2023)).

Table 1: **LLaDA-8B-Instruct.** Math/code use 512 generation slots; IFEval uses 2,048. Scores are percentages; NFE is mean model calls/request and TPS is effective tokens/s. Bold marks the best displayed value per task and metric, including ties; – denotes an unreported result.

Method	GSM8K (5 − *shot*)			MATH500 (4 − *shot*)			HumanEval (0 − *shot*)			MBPP (3 − *shot*)			IFEval (0 − *shot*)		
	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑	Strict ↑	NFE ↓	TPS ↑
Original	76.70	512.00	7.71	40.80	512.00	16.26	36.34	512.00	21.88	37.80	512.00	4.84	50.38	2048.00	3.45
Fast-dLLM	74.50	102.41	65.76	40.80	168.11	69.57	39.63	114.33	95.05	35.60	73.03	84.63	50.83	264.35	44.49
AdaBlock	78.50	108.04	57.58	**41.40**	177.76	67.23	39.63	126.97	86.06	37.00	67.71	58.01	51.02	284.29	39.48
AHD	**81.20**	89.95	38.13	**41.40**	151.03	46.88	43.29	102.85	84.46	**38.00**	84.11	57.48	**56.75**	207.58	10.62
KLASS	77.94	118.64	28.61	40.00	200.56	35.68	**45.12**	177.03	54.81	35.60	77.69	30.84	54.53	304.21	17.98
Ours	79.68	**59.00**	**101.83**	39.90	**105.94**	**104.10**	39.41	**77.66**	**132.96**	36.40	**47.36**	**129.23**	51.94	**176.41**	**72.75**
vs. AdaBlock	(+1.18)	1.83×	1.77×	(−1.50)	1.68×	1.55×	(−0.22)	1.63×	1.55×	(−0.60)	1.43×	2.23×	(+0.92)	1.61×	1.84×

Table 2: **Dream-v0-Instruct-7B.** Budgets and notation follow Table 1. AHD is our Dream port of the official LLaDA scheduler.

Method	GSM8K (5 − *shot*)			MATH500 (4 − *shot*)			HumanEval (0 − *shot*)			MBPP (3 − *shot*)			IFEval (0 − *shot*)		
	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑	Strict ↑	NFE ↓	TPS ↑
Original	76.88	512.00	4.35	45.40	512.00	9.43	**58.76**	512.00	12.18	53.00	512.00	2.24	47.97	2048.00	2.84
Fast-dLLM	78.09	79.25	46.69	44.60	127.43	63.76	57.93	103.86	62.06	53.60	58.25	28.39	46.40	297.97	35.68
AdaBlock	77.56	80.42	48.06	43.20	128.27	59.33	58.54	107.13	58.92	53.40	61.53	25.86	47.50	307.15	36.45
AHD (port)	77.86	62.51	31.72	46.60	105.75	35.49	53.05	78.02	53.56	56.60	41.18	25.09	53.60	175.45	8.84
KLASS	78.09	107.83	17.00	43.20	132.76	20.76	51.83	116.98	35.15	**59.80**	57.22	16.71	**55.82**	**124.42**	8.50
Ours	**79.87** ±0.36	**57.15**	**67.20**	**47.00**	**84.68**	**91.28**	57.93	**64.38**	**95.25**	56.00	**28.46**	**39.88**	46.58	145.13	**85.65**
vs. AdaBlock	(+2.31)	1.41×	1.40×	(+3.80)	1.51×	1.54×	(−0.61)	1.66×	1.62×	(+2.60)	2.16×	1.54×	(−0.92)	2.12×	2.35×

Baselines. LLaDA and Dream baselines are Original (native decoding without KV caching), Fast-dLLM (Wu et al., 2025) (approximate KV caching), AdaBlock-dLLM (Lu et al., 2025) (adaptive boundaries atop Fast-dLLM), AHD (Zou et al., 2026), and KLASS (Kim et al., 2025). Dream AHD ports the official LLaDA scheduler with Dream's native logit alignment. DiffusionGemma uses its official entropy-bound decoder as baseline. Methods share weights and prompt formats within each model.

Metrics and timing. We report math accuracy, code pass@1, and IFEval prompt-level strict accuracy, alongside forward call count (NFE) and effective output throughput (TPS: generated tokens before EOS divided by request time). DiffusionGemma NFE counts encoder and decoder calls; TPS includes thinking tokens and excludes padding. Timing uses a single NVIDIA H100 80GB HBM3, batch size 1, exclusive device access, and a warmup pass. TPS ratios compare throughput, not controlled latency speedups.

Configuration. LLaDA and Dream use 512-token math/code canvases and 2,048 tokens for IFEval; DiffusionGemma uses 1,536 tokens on GSM8K and 8,192 elsewhere. All runs use bfloat16. AHD and KLASS use one seed per task: 42 for math/code and 0 for IFEval. AsyncLane's LLaDA MBPP and Dream GSM8K, MATH500, and MBPP results average seeds 45–48. Configurations are selected per task, sometimes using evaluation examples. Dream KLASS uses temperature 0.2 and top-p 0.95; AsyncLane uses temperature 0.3 on Dream GSM8K, 0.2 on MBPP, and 0 on other Dream tasks. Comparisons concern these configurations and timing batches; Appendix E gives further settings and defaults.

4.2 Task Quality and Inference Efficiency

Throughput gains across models. AsyncLane achieves the highest TPS in every evaluated model–task combination within the main comparison budgets (Tables 1–3). Its TPS reaches 2.40× that of Fast-dLLM on Dream IFEval. AsyncLane also uses the fewest model calls except on Dream IFEval.

Table 3: **DiffusionGemma-26B-A4B-it.** Notation and boldface follow Table 1.

Method	GSM8K (0 − *shot*)			MATH500 (0 − *shot*)			HumanEval (0 − *shot*)			MBPP (0 − *shot*)			IFEval (0 − *shot*)		
	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑	Strict ↑	NFE ↓	TPS ↑
Original	85.14	25.97	296.10	**94.00**	85.24	276.93	**95.12**	59.57	310.97	91.00	89.88	251.39	91.68	85.67	223.02
Ours	**85.44**	**22.04**	**337.64**	93.00	**79.19**	**283.76**	92.68	**51.10**	**313.22**	**91.60**	**73.85**	**293.44**	**92.05**	**74.84**	**274.44**
vs. Original	(+0.30)	1.18×	1.14×	(−1.00)	1.08×	1.02×	(−2.44)	1.17×	1.01×	(+0.60)	1.22×	1.17×	(+0.37)	1.14×	1.23×

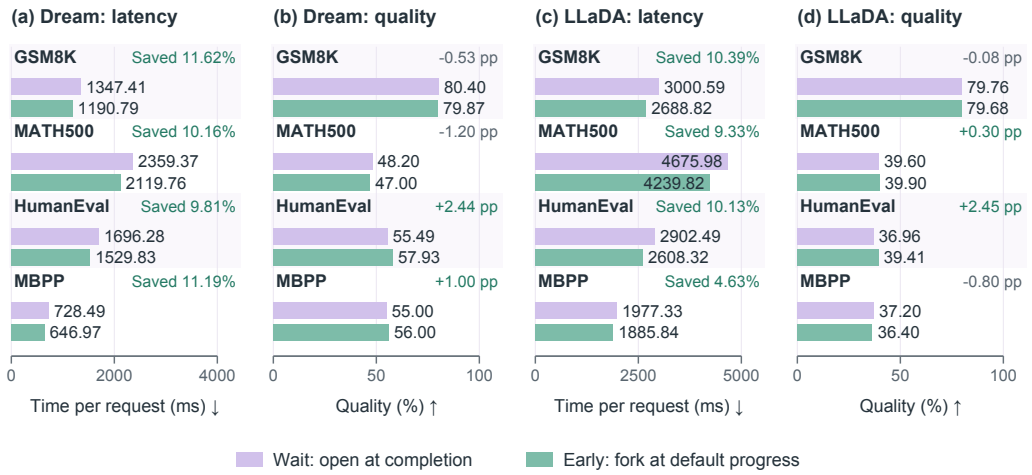


Figure 4: Early versus completion-based opening. Bars show mean request latency and task quality. Saved (%) is the latency reduction relative to Wait; quality differences are Early minus Wait in percentage points. Latency scales differ. Table 8 reports paired latency intervals; quality is reported as point estimates.

Joint gains in quality and throughput. On Dream, AsyncLane achieves the highest accuracy on both math benchmarks while using the fewest model calls. It also improves both quality and throughput over Fast-dLLM and AdaBlock on MBPP, and over AHD on GSM8K, MATH500, and HumanEval under the same generation budget. On LLaDA, AsyncLane improves both metrics over Fast-dLLM on GSM8K and MBPP, and over AdaBlock on GSM8K.

Efficient instruction following. On IFEval, AsyncLane delivers the highest TPS on both models, with higher strict accuracy than Fast-dLLM and, on LLaDA, AdaBlock. On Dream, it exceeds KLASS in TPS despite using more calls. Call costs and effective output lengths can differ, so TPS ratios are not directly request-latency speedups. Quality gains remain task dependent, with some baselines scoring higher on IFEval.

Gains on an encoder–decoder model. AsyncLane reduces NFE and increases TPS on every DiffusionGemma task, while improving quality on GSM8K, MBPP, and IFEval. The adaptation overlaps adjacent canvases and updates context at handoff (Appendix D). HumanEval shows only a small throughput improvement with lower pass@1. Section 4.3 examines early opening and prediction reuse through controlled comparisons.

4.3 Ablation Studies

We use a common set of ablation settings on LLaDA and Dream to examine the latency contributions of early opening and full lookahead prefill (LA). These settings differ from the task-specific configurations in the main results. Figures 4 and 5 report mean request latency alongside task quality; the appendices provide the corresponding point estimates and request-paired 95% latency intervals.

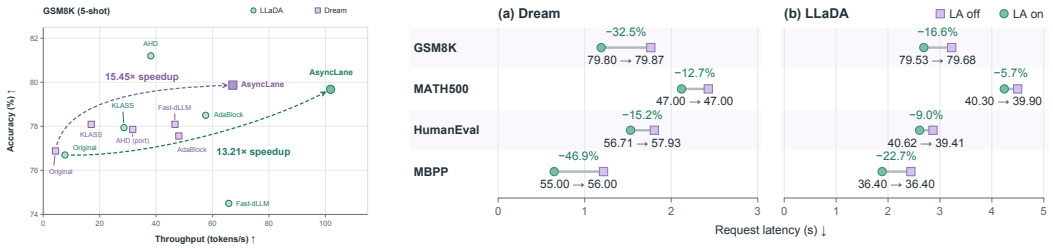


Figure 5: **Accuracy–throughput trade-off and full LA ablation. Left:** GSM8K (5-shot) accuracy versus throughput for LLaDA (green) and Dream (purple). Dashed arrows connect Original to AsyncLane; labels give the corresponding throughput ratios. **Right:** LA off (purple) versus LA on (green) under fixed early opening. Markers show mean request latency; percentages give changes relative to LA off (negative means faster). Quality (%) is shown as off → on. The two latency panels use different axis ranges. All results are point estimates; paired latency intervals are reported in Table 9.

Early opening accumulates progress before handoff. We compare early opening ($\rho = 0.7$) with waiting for the current block to finish ($\rho = 1$), keeping LA, block partitioning, and candidate admission fixed. In Figure 4, early opening lowers mean request latency on every tested task, with the largest reduction of 11.62% on Dream GSM8K. This pattern is consistent with the benefit of accumulating candidates early: accepted private candidates can be merged at handoff, leaving fewer positions for the next block to fill. The comparison supports a practical time benefit from moving some advancement before current-block completion. LLaDA MBPP is the main uncertainty: its estimated benefit is smaller, and its latency interval includes zero.

Quality changes depend on the task. Dream's code scores improve while its math scores decline; LLaDA likewise shows both gains and losses. Candidate admission rules restrict which predictions enter the private lane, but do not guarantee that changing the opening schedule preserves task quality. These results therefore support a latency benefit from early opening while retaining its task-dependent quality trade-offs.

Reusing opening-forward predictions can reduce subsequent work. We compare full LA with LA disabled while fixing early opening ($\rho = 0.7$), candidate admission, and block partitioning. Disabling LA retains the private lane's early advancement, so this contrast examines the additional contribution of prediction reuse. In Figure 5, full LA lowers mean request latency in every tested comparison, reaching a reduction of 46.85% on Dream MBPP; all latency intervals support a reduction. LA uses predictions already computed by the opening forward. Content prefill can fill positions before their regions open, while EOS padding can reduce later block openings and decoding beyond the retained output. These operations provide ways to reduce subsequent work without a dedicated forward pass to obtain the predictions.

The quality results distinguish time savings from quality gains. Dream MATH500 retains the same accuracy estimate while request latency decreases, whereas LLaDA MATH500 and HumanEval have lower quality estimates with LA enabled. The intervention includes content prefill, EOS padding, and associated skipping, so the observed benefit belongs to full LA and cannot be attributed to content prefill alone. The two ablations support the time benefits of advancing earlier and reusing existing predictions under their respective controls. Their reductions use different references and do not form an additive decomposition of computation. Appendix H examines further trade-offs in the opening threshold and content prefill horizon.

5 Conclusion

AsyncLane accumulates next-block progress in a private lane under incomplete context, with handoff after current-block completion. Across three models, it improves effective output throughput under the tested configurations, with quality gains on multiple tasks. Controlled LLaDA and Dream ablations support request-latency benefits from early opening and an additional reduction from full LA under fixed opening.

AI USE STATEMENT

In this work, we used generative AI tools for writing assistance (language editing of author-written text) and for coding assistance (implementation and refactoring of experiment scripts). We have not used generative AI tools for research ideation, experimental design, or analysis of results. We have reviewed all AI-assisted work: all edited text was checked by the authors for fidelity to the intended claims, and all AI-assisted code was tested and its outputs verified against independently produced results.

ETHICS STATEMENT

This work evaluates AsyncLane on public language and code benchmarks and involves no human-subject studies or collection of personal data. AsyncLane improves the inference efficiency of diffusion language models through decoding-time scheduling, without additional training or changes to backbone parameters. Its broader societal considerations are those common to language-model research.

REPRODUCIBILITY STATEMENT

The method section specifies AsyncLane's decoding procedure, early-opening criterion, private-lane updates, and rolling handoff. Evaluation protocols and inference hyperparameters are detailed in the experimental setup section. All benchmarks and pretrained backbones used in our experiments are publicly available, and AsyncLane requires no additional model training. Code, inference and evaluation scripts, and experiment configurations will be released upon publication.

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, 2025. URL https://arxiv.org/abs/2503.09573.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

Wenrui Bao, Zhiben Chen, Dan Xu, and Yuzhang Shang. Learning to parallel: Accelerating diffusion large language models via learnable parallel decoding, 2025. URL https://arxiv.org/abs/2509.25188.

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

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, 2021. URL https://arxiv.org/abs/2110.14168.

DiffusionGemma Team, Adrien Ali Taïga, James Assiene, Daniele Calandriello, Rahma Chaabouni, João Gante, Tamara von Glehn, Nate Keating, Chris Knutsen, Martin Kukla, Tianlin Liu, Ivan Lobov, Ofir Nabati, João Gabriel Oliveira, Nicolas Perez-Nieves, Nastasia Prutianova, Bobak Shahriari, Jean Tarbouriech, Pavel Tyletski, Çağlar Ünlü, Cindy Wu, Glenn Cameron, Jerome

Connor, Sertan Girgin, Maarten Grootendorst, Alon Levkovitch, Eliya Nachmani, Omar Sanseviero, Piotr Stanczyk, Quentin Berthet, Andrew Campbell, Clément Crepy, Valentin De Bortoli, Arnaud Doucet, Romuald Elie, Alexandre Galashov, Klaus Greff, Alexis Jacq, David Ruhe, Yu-Han Wu, Sebastian Flennerhag, Brendan O'Donoghue, George Scrivener, and Shantanu Thakoor. DiffusionGemma technical report, 2026. URL https://arxiv.org/abs/2608.00146.

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, 2021. URL https://arxiv.org/abs/2103.03874.

Daniel Israel, Guy Van den Broeck, and Aditya Grover. Accelerating diffusion llms via adaptive parallel decoding, 2025. URL https://arxiv.org/abs/2506.00413.

Seo Hyun Kim, Sunwoo Hong, Hojung Jung, Youngrok Park, and Se-Young Yun. KLASS: KL-guided fast inference in masked diffusion models. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 92267–92301. Curran Associates, Inc., 2025. doi: 10.52202/085713-3087. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/8514c1f12dcdd061ac8521feb78f28ca-Paper-Conference.pdf.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step, 2023. URL https://arxiv.org/abs/2305.20050.

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, 2026. URL https://arxiv.org/abs/2506.06295.

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, 2025. URL https://arxiv.org/abs/2509.26432.

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, 2025. URL https://arxiv.org/abs/2502.09992.

Tianxiang Pan, Baitao Gong, Mo Guang, Hongwei Yong, Tianpeng Jiang, Yaqian Li, Zheng Cao, and Kaiwen Long. Trajectory-level speculative decoding for diffusion language models, 2026. URL https://arxiv.org/abs/2608.27514.

Bing Tian, Haikun Liu, Xiaocheng Zhong, Zhuohui Duan, Zhaokai Luo, Huayi Jin, Zhiyong Wang, and Xiaofei Liao. Flowblock: Wavefront-parallel decoding for self-correcting diffusion language models, 2026. URL https://arxiv.org/abs/2607.17652.

Xu Wang, Chenkai Xu, Yijie Jin, Jiachun Jin, Hao Zhang, and Zhijie Deng. Diffusion llms can do faster-than-ar inference via discrete diffusion forcing, 2025. URL https://arxiv.org/abs/2508.09192.

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, 2025. URL https://arxiv.org/abs/2505.22618.

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

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models, 2023. URL https://arxiv.org/abs/2311.07911.

Shun Zou, Yong Wang, Zehui Chen, Lin Chen, Chongyang Tao, Feng Zhao, and Xiangxiang Chu. Breaking block boundaries: Anchor-based history-stable decoding for diffusion large language models. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 20637–20658, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.945. URL https://aclanthology.org/2026.acl-long.945/.

A EOS PROCESSING RULES

This section supplements the EOS-related rules beyond Section 3.4: how the two lanes write EOS, the complete conditions for EOS padding, the relationship between the first EOS and prefill records, and when decoding stops and how the output is truncated.

Two lanes. The main lane commits EOS by the rule of Section 3.1, the same as any other token. The private lane may accept EOS as a candidate by the acceptance rule of Section 3.3; once merged, it enters the main state in the same way.

EOS padding conditions. Compared with content prefill, EOS padding differs in four ways: it is performed only after an EOS already exists in the main state, and only at positions after the first EOS; it uses the main-lane commit threshold τ_{main}; it is not limited by H; and writes are neither recorded nor rescored. Let e be the position of the first EOS in the main state. EOS padding writes to every position that satisfies all of the following:

$$x_i = \texttt{MASK}, \qquad i \geq r, \qquad i > e, \qquad \hat{y}_i = \texttt{EOS}, \qquad q_i \geq \tau_{\text{main}}. \tag{5}$$

Here $\hat{y}_i$, q_i, and r are the same as in Section 3.4. These positions are all past the first EOS and do not belong to the output; filling them early prevents the decoder from opening new blocks for this part of the canvas. The value of e is determined from the main state after refining A within the current opening forward; EOS tokens among private candidates are not counted until they are merged.

First EOS and prefill records. Content prefill does not write EOS, and EOS padding requires an EOS already in the main state, so the first EOS in the main state can only be committed by the main lane or written by candidate merge—always inside an opened block, i.e., $e < r$. Consequently, once the main state contains an EOS, all positions written by lookahead prefill are past e; these positions, including any content prefill among them, are not added to the prefill record and are not rescored. Content prefill recorded before the first EOS is still rescored per Section 3.4, until it is either revoked or removed from the record by falling left of r.

Stopping and output. The decoder does not stop upon encountering an EOS. Positions written by EOS padding, like those written by content prefill, are skipped when determining the start of B (Section 3.2). When no `MASK` remains after the current block—either because all subsequent positions have been filled by lookahead prefill, or because the current block extends to the canvas end—the opening condition is no longer satisfied. The decoder then only refines the current block; decoding ends when the current block is complete, at which point the entire canvas is filled. The default output processing retains tokens before the first EOS; if no EOS was generated, the entire canvas is retained. Special tokens are skipped when converting to text; task-specific post-processing is described in the experimental setup.

B CACHE AND MODEL-SPECIFIC IMPLEMENTATION DETAILS

This section supplements the execution details of Section 3.5: the three call types, the active interval, the two-row cache and handoff, when writes are re-encoded, and model-specific implementation details.

Three call types. Model calls during decoding fall into three types. The opening forward takes the main state as a single row and computes all positions; the initial candidates for B are also drawn from its output. Overlap calls occur after opening B and before A is complete, taking the main state and the private view as two rows. Refinement calls occur when no private lane is active, taking the main state as a single row. The latter two are collectively called cached calls and compute only the active interval.

Approximate KV cache. The cache follows the DualCache design of Fast-dLLM (Wu et al., 2025). The opening forward recomputes all positions and replaces the entire cache; cached calls replace only the keys and values within the active interval, reusing the cache at positions outside it.

Active interval. The active interval for each cached call is $[s, r)$, where s is the start of the current block at the most recent opening forward and r is as in Section 3.4. During overlap, the active interval is $[a, d)$, covering both A and B. Since s is updated only at opening forwards, the just-completed block remains inside the active interval after handoff until the next opening forward.

Two-row cache and handoff. The cache produced by the opening forward is duplicated into two copies, one per lane; during overlap calls, each lane replaces only its own copy within the active interval. At handoff, the main lane's cache is kept and the private lane's cache is discarded: outside the active interval, both copies originate from the same opening forward and are identical; inside the interval, the entries will be recomputed in the next call, so the choice does not affect subsequent computation.

When writes are re-encoded. Writes within the active interval—including main-lane commits and merged candidates—are encoded with their new content at the next call. Lookahead prefill writes and revocations update the main state immediately but do not refresh the cache at the corresponding positions. These positions lie outside the active interval, so subsequent cached calls still reuse the keys and values computed by the current opening forward, until the next opening forward recomputes them from the updated main state.

Accordingly, the statement in Section 3.3 that "lookahead prefill in the main state appears in both lanes' inputs" refers, under caching, to prefill written before the current opening forward and already encoded into the cache by it. The rescoring of Section 3.4 takes place at a later opening forward, at which point the input at that position is the prefilled token itself.

Dream: output alignment and predecessor query. In Dream, the output at position $i - 1$ gives the prediction for position i. The Dream implementation released by AdaBlock-dLLM shifts the output right by one position before use; we follow this convention. That implementation force-writes the first token of each block at opening, so cached calls do not need a prediction for the block-start position. AsyncLane does not force-write the block start; after opening or after merge, the block-start position may still be `MASK`. To handle this, every cached call in Dream begins its query at $s-1$, using that position's output to obtain the prediction for position s and recomputing its keys and values along with the rest; when $s = 0$, $s-1$ is the last token of the prompt. Consequently, in Dream the range queried and replaced by cached calls is $[s - 1, r)$; the cache descriptions in this section should be understood accordingly.

C Complete Decoding Algorithm

Algorithm 2 expands Algorithm 1 with cache management and the active query interval. The acceptance and prefill rules follow Sections 3.1–3.4; EOS handling and model-specific query alignment are specified in Appendices A and B.

Algorithm 2 AsyncLane decoding loop. MainUpdate, Boundary, Admit and Lookahead apply the rules of Sections 3.1, 3.2, 3.3 and 3.4, respectively. An opening forward evaluates the full sequence with one row, the main state, and refreshes the cache $\mathcal{K}$. Overlap calls (two rows) and refinement calls (one row) query the active interval $[s, r)$ and reuse cached keys and values at the other positions. Lookahead changes only positions at or beyond r, which lie outside these queries, so its writes and revocations are re-encoded at the next opening forward. Appendix A gives the EOS rules; Appendix B gives the exact query range, the cache handling at handoff and model-specific details.

Require: Frozen model, prompt, canvas length L; main-lane commit threshold τ_{main}, opening threshold ρ, private acceptance threshold τ_{priv}, fallback floor τ_{floor}, delimiter threshold τ_{bd}, fallback block length ℓ_0, prefill threshold τ_{LA}, lookahead horizon H

$x_i \leftarrow$ `MASK` for $i = 0, \ldots, L-1$; prefill records $T \leftarrow \varnothing$ {x: main state; every model row is the prompt followed by a canvas}
$(P, \mathcal{K}) \leftarrow \text{OpeningForward}(x)$ {Opens the first block; P: predictions at all positions}
$a \leftarrow 0$; $b \leftarrow \text{Boundary}(a, P)$; $A \leftarrow [a, b)$ {Delimiter-confidence rule of AdaBlock-dLLM; b is the exclusive end}
$x \leftarrow \text{MainUpdate}(x, A, P)$ {Main-lane rule; the first block opens without a private lane}
$r \leftarrow b$; $(x, T) \leftarrow \text{Lookahead}(x, T, P, r)$ {Content prefill, Eq. 4, and EOS padding}
$s \leftarrow a$ {s: start of the current block at the latest opening forward}
loop
  $g \leftarrow 1 - m(A, x)/|A|$ {Fill ratio $g(A, x)$, Eq. 1}
  if $g \geq \rho$ **and** a `MASK` remains after A **then**
    $(P, \mathcal{K}) \leftarrow \text{OpeningForward}(x)$ {Opens B}
    $x \leftarrow \text{MainUpdate}(x, A, P)$ {Writes nothing if $m(A, x) = 0$}
    $c \leftarrow$ first `MASK` position $\geq b$ {Skips prefilled positions right after A}
    $d \leftarrow \text{Boundary}(c, P)$; $B \leftarrow [c, d)$; $r \leftarrow d$; $s \leftarrow a$
    $(x, T) \leftarrow \text{Lookahead}(x, T, P, r)$ {Drops records left of r, rescores the rest, then writes}
    $C \leftarrow \text{Admit}(\varnothing, x, B, P)$ {Initial candidates from the same output}
    $\mathcal{K}^{(2)} \leftarrow (\mathcal{K}, \mathcal{K})$ {One cache row per lane}
    while $m(A, x) > 0$ **do**
      $\widetilde{x} \leftarrow$ private view of x and C {Eq. 2}
      $(P^{\text{main}}, P^{\text{priv}}, \mathcal{K}^{(2)}) \leftarrow \text{OverlapCall}(x, \widetilde{x}, [s, r), \mathcal{K}^{(2)})$
      $x \leftarrow \text{MainUpdate}(x, A, P^{\text{main}})$
      $C \leftarrow \text{Admit}(C, x, B, P^{\text{priv}})$ {Over U_B, Eq. 3}
    end while
    $x_i \leftarrow C_i$ for all $i \in \text{dom}(C)$ with $x_i =$ `MASK` {Handoff; at once if A is complete after the opening forward}
    $\mathcal{K} \leftarrow$ main-lane row of $\mathcal{K}^{(2)}$; discard C {The private lane ends}
    $A \leftarrow B$, i.e., $(a, b) \leftarrow (c, d)$ {s and r unchanged; the opening check follows at once}
  else if $m(A, x) > 0$ **then**
    $(P, \mathcal{K}) \leftarrow \text{RefinementCall}(x, [s, r), \mathcal{K})$ {One row}
    $x \leftarrow \text{MainUpdate}(x, A, P)$
  else
    break {A is complete and no `MASK` remains after it}
  end if
end loop
return the text of x up to its first EOS

D ADAPTING ASYNCLANE TO DIFFUSIONGEMMA

Model differences. In our implementations for LLaDA and Dream, decoding proceeds by filling `MASK` positions, and private progress is retained as candidate tokens. DiffusionGemma uses an encoder–decoder architecture: the encoder provides context representations of the finalized prefix, while the decoder iteratively denoises a canvas of fixed length. To accommodate this difference, we adapt AsyncLane's state representation and handoff procedure, applying its scheduling principles of early opening, private advancement, and post-completion handoff to adjacent canvases.

Adapting the schedule. We use agreement between the main canvas's argmax predictions at two consecutive steps over at least 90% of positions as the signal for opening the next canvas early. The main canvas continues denoising while the next canvas independently accumulates denoising progress as the private lane. The two canvases advance as two batch rows in a single decoder call. Before the main canvas is complete, the private state is neither committed nor used as input to the main canvas. This allows computation on the next canvas to begin before the current one is complete while preserving the order of output commits.

Context handoff. The private canvas uses the main canvas's decoder K/V from the opening step as temporary prefix context, which remains fixed during overlap. Once the main canvas is complete, if further generation is needed, we encode its finalized tokens once and replace the temporary K/V with the resulting encoder context. The private canvas then becomes the new main canvas. Handoff preserves its private denoising state and self-conditioning information. We perform at least one further decoder forward pass using this context to update the canvas before checking again whether it is complete.

E EXPERIMENTAL CONFIGURATION

E.1 SHARED EVALUATION SETTINGS

All methods use frozen model weights and bfloat16 precision. We evaluate on a single NVIDIA H100 80GB HBM3 with a request batch size of one and warm up the model before timing. During overlap, AsyncLane's two lanes form two input rows for the same request.

Table 4: **Task sizes, few-shot counts, and output budgets.** Budgets exclude input prompts and include thinking tokens for DiffusionGemma; its canvas length is given in Table 7.

		LLaDA / Dream		**DiffusionGemma**	
Task	Items	Shots	Budget	Shots	Budget
GSM8K	1,319	5	512	0	1,536
MATH500	500	4	512	0	8,192
HumanEval	164	0	512	0	8,192
MBPP	500	3	512	0	8,192
IFEval	541	0	2,048	0	8,192

For LLaDA and Dream, GSM8K uses five fixed training examples, MATH500 uses four fixed Minerva examples, and MBPP uses three examples from the official prompt split. Methods share the same prompt construction within each model and task. DiffusionGemma has thinking enabled on all tasks.

E.2 LLADA AND DREAM DECODING SETTINGS

We denote the main-lane commit threshold by τ_{main} and the early-opening progress threshold by ρ. The private-candidate threshold and fallback acceptance floor are τ_{priv} and τ_{floor}, respectively. The delimiter-confidence threshold is τ_{bd}, and ℓ_0 is the initial and fallback block length. For content prefill, τ_{LA} is the confidence threshold and H is the horizon. All lengths are in tokens.

Table 5: **Baseline decoding settings.** Original follows each model's native denoising schedule; call counts for Fast-dLLM and AdaBlock-dLLM are determined by threshold-based decoding. LA denotes lookahead prefill.

Model	Method	Temp. / top-p	Blocks	Commit	Boundary	KV cache	LA
LLaDA	Original	0 / —	Fixed 32	—	—	None	Off
Dream	Original	0.1 / 0.9	Full canvas	—	—	None	Off
Both	Fast-dLLM	0 / —	Fixed 32	0.9	—	Dual	Off
LLaDA	AdaBlock-dLLM	0 / —	Adaptive, 32	0.9	0.3	Dual	Off
Dream	AdaBlock-dLLM (math/code)	0 / —	Adaptive, 32	0.9	0.3	Dual	Off
Dream	AdaBlock-dLLM (IFEval)	0 / —	Adaptive, 32	0.9	0.5	Dual	Off

For both models, Original executes 512 steps on math and code tasks and 2,048 steps on IFEval. In Table 5, adaptive blocks use an initial and fallback length of 32. Dual caching reuses approximate K/V on both sides of the current query interval; AsyncLane uses the same cache mode. The three baseline methods in the table use a base seed of 42 on math and code tasks and 0 on IFEval.

Table 6: **Default hyperparameters of AsyncLane.**

Model	τ_{main}	ρ	τ_{priv}	τ_{floor}	τ_{bd}	ℓ_0
LLaDA	0.7	0.7	0.9	0.5	0.3	32
Dream	0.7	0.7	0.9	0.5	0.5	32

Block boundaries and prefill. For AdaBlock-dLLM and AsyncLane, the delimiter search window is capped at one quarter of the output canvas length and clipped at the canvas end. The cap is therefore 128 tokens for a 512-token canvas and 512 tokens for a 2,048-token canvas. LLaDA uses delimiter token ID 198; Dream uses the following 24 IDs:

```
198, 271, 280, 319, 340, 382, 401, 532, 624, 630, 692, 698,
921, 1248, 1837, 1939, 2219, 2533, 3276, 3876, 4894, 5267,
14750, 68327.
```

EOS-only mode does not use the content-prefill threshold τ_{LA} or horizon H. All listed AsyncLane configurations use τ_{main} as the confidence threshold for EOS padding.

E.3 DIFFUSIONGEMMA DECODING SETTINGS

Table 7 compares the official decoder with AsyncLane. These settings are shared across all five tasks, with output budgets specified in Table 4. The adaptation procedure is described in Appendix D.

Table 7: **DiffusionGemma decoding settings.**

Setting	Original (official decoder)	AsyncLane
Canvas length	256 tokens	256 tokens
Base denoising schedule	48 steps	48 steps
Temperature	Linear annealing from 0.8 toward 0.4	Same schedule, using each canvas's remaining steps
Sampler / entropy budget	EntropyBoundSampler / 0.1	Same
Canvas early stopping	Mean predictive entropy < 0.005 and identical whole-canvas argmax predictions at two consecutive steps	Mean predictive entropy < 0.005
Early opening	Disabled	Main-canvas argmax agreement ≥ 0.9 across positions at two consecutive steps
Lookahead prefill	Off	Off
Random seed	1234 + zero-based item index in the original task dataset	Same

F Early Opening: Detailed Results

This appendix accompanies the eight early-versus-completion-based opening comparisons in Figure 4. Both schedules use the same LA, block partitioning, and candidate admission rules. These ablation settings differ from the task-specific configurations in the main results. Table 8 reports latency reductions with paired intervals and quality differences as point estimates.

Latency first averages three timings for each request, then averages over requests within a task. Saved latency is $100 \times (T_{\mathrm{Wait}} - T_{\mathrm{Early}})/T_{\mathrm{Wait}}$; positive values favor Early. Quality is accuracy for the math tasks and pass@1 for the code tasks. Quality differences are Early minus Wait in percentage points. Latency intervals retain the existing request-paired, pointwise 95% bootstrap estimates without multiplicity correction. Latency effects and intervals use the existing unrounded timing statistics; quality differences are calculated from the scores displayed in the revised figure.

Table 8: Early versus Wait under fixed LA and partitioning (Figure 4). Savings are relative to Wait. Quality differences are Early minus Wait in percentage points. Brackets report request-paired, pointwise 95% latency intervals; quality differences are point estimates from the displayed scores.

Model	Task	Saved latency (%) [95% CI]	Δ quality (pp)
Dream	GSM8K	11.62 [10.56, 12.60]	−0.53
Dream	MATH500	10.16 [7.30, 12.88]	−1.20
Dream	HumanEval	9.81 [3.23, 15.62]	2.44
Dream	MBPP	11.19 [9.48, 12.90]	1.00
LLaDA	GSM8K	10.39 [9.20, 11.62]	−0.08
LLaDA	MATH500	9.33 [7.65, 10.95]	0.30
LLaDA	HumanEval	10.13 [7.40, 12.74]	2.45
LLaDA	MBPP	4.63 [−0.60, 9.38]	−0.80

The latency-reduction interval includes zero for LLaDA MBPP and is above zero for the other seven comparisons. Quality point estimates show both gains and losses, so the timing benefit does not establish quality equivalence or lossless behavior.

G Full LA Ablation

This appendix accompanies the full-LA versus LA-disabled comparison in Figure 5. Both configurations retain early opening with fixed candidate admission and block partitioning. Full LA includes content prefill, EOS padding, and associated skipping. Table 9 reports absolute request latency, latency reductions with paired intervals, and quality point estimates.

Latency first averages three timings for each request, then averages over requests within a task. Saved latency is $100 \times (T_{\mathrm{off}} - T_{\mathrm{on}})/T_{\mathrm{off}}$, and quality differences are on minus off in percentage points. Latency intervals retain the existing request-paired, pointwise 95% bootstrap estimates without multiplicity correction; their endpoints are transformed from the existing latency-ratio intervals without resampling. Latency statistics use the existing unrounded values, while quality differences are calculated from the scores displayed in the revised figure.

All eight latency-reduction intervals are above zero. Quality changes depend on the task: Dream MATH500 has the same accuracy estimate, while LLaDA MATH500 and HumanEval decline. These comparisons support a practical time benefit from full LA, but do not attribute the entire benefit to content prefill or establish quality equivalence.

Table 9: Full LA versus LA disabled under fixed early opening: (a) request latency with paired intervals and (b) task quality point estimates.

(a) Request latency

Model	Task	LA off (ms)	LA on (ms)	Saved (%)	95% CI (%)
Dream	GSM8K	1764.16	1190.79	32.50	[31.85, 33.18]
Dream	MATH500	2428.49	2119.76	12.71	[11.61, 13.95]
Dream	HumanEval	1804.99	1529.83	15.24	[11.76, 18.50]
Dream	MBPP	1217.36	646.97	46.85	[45.56, 48.20]
LLaDA	GSM8K	3225.14	2688.82	16.63	[15.93, 17.36]
LLaDA	MATH500	4497.31	4239.82	5.73	[4.74, 6.69]
LLaDA	HumanEval	2866.24	2608.32	9.00	[7.15, 10.91]
LLaDA	MBPP	2440.83	1885.84	22.74	[20.70, 24.72]

(b) Task quality

Model	Task	LA off (%)	LA on (%)	Change (pp)
Dream	GSM8K	79.80	79.87	0.07
Dream	MATH500	47.00	47.00	0.00
Dream	HumanEval	56.71	57.93	1.22
Dream	MBPP	55.00	56.00	1.00
LLaDA	GSM8K	79.53	79.68	0.15
LLaDA	MATH500	40.30	39.90	−0.40
LLaDA	HumanEval	40.62	39.41	−1.21
LLaDA	MBPP	36.40	36.40	0.00

Notes. Savings are relative to LA off; quality change is LA on minus LA off. Latency intervals are request-paired and pointwise. Latency changes use the existing unrounded statistics; quality differences use the displayed scores.

H PARAMETER SENSITIVITY

We examine the opening threshold ρ and the content prefill horizon H, which control when advancement begins and how far predictions may be written directly to the main state. We evaluate LLaDA and Dream on MATH500 and MBPP, with 500 requests per task and a 512-position generation canvas. We vary one parameter at a time, using the $\rho = 0.7$, $H = 32$ configuration from the same parameter study as the reference. All other candidate acceptance and prefill rules remain fixed.

We report MATH500 accuracy and MBPP pass@1 from the first run. Request latency is averaged over three timed runs per request, then over requests within each task. The figures show quality differences and latency changes relative to the reference; a positive latency change means slower decoding. Error bars are pointwise 95% bootstrap intervals paired by request, without correction for multiple comparisons.

H.1 OPENING THRESHOLD

The main-text instantiation defines the fill ratio in Equation equation 1; A lower threshold starts the attempt to advance the next block earlier; candidate acceptance remains governed by its separate rule. We compare $\rho \in \{0.5, 0.7, 0.9, 1.0\}$, where $\rho = 1$ waits for the current block to complete before opening the next.

Later opening increases mean request latency in Figure 6. Raising the threshold from the default to 0.9 or 1.0 increases mean latency for both models on both tasks. Relative to waiting for completion, the default reduces mean latency by approximately 5%–11%. This trend is consistent with the schedule: opening near current-block completion leaves less overlap for private candidates to accumulate, potentially reducing the work completed before handoff.

Opening still earlier does not always yield further gains. At $\rho = 0.5$, Dream is faster on MBPP but shows no additional latency benefit on MATH500. Execution records show fewer model calls (NFE) but more total input rows processed in all four model–task combinations. The saved calls therefore need to be considered alongside the cost of two-row computation. For example, if no private prediction passes the acceptance rule in a given call, the private input still incurs computation without adding a candidate for handoff. These records suggest that the net benefit of earlier opening depends on whether accumulated progress offsets the extra computation; lower NFE alone does not establish lower request latency.

Quality scores show no consistent direction of change across tasks as the threshold varies. Every quality-difference interval between a nondefault setting and the default includes zero. Later opening provides no consistent quality benefit, while earlier opening does not further accelerate every task. We therefore retain $\rho = 0.7$ as the default operating point, attempting overlap after the current block has made progress. The threshold remains available to adjust the quality–latency tradeoff for a particular task.

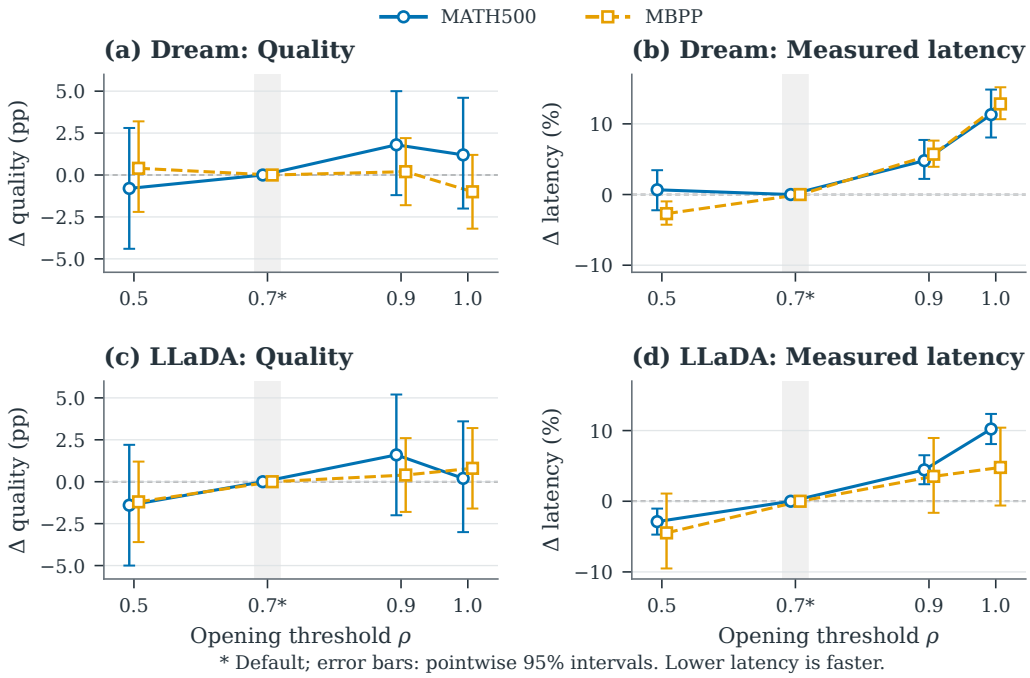


Figure 6: **Opening threshold sensitivity.** The top and bottom rows show Dream and LLaDA. The left column reports quality differences in percentage points relative to $\rho = 0.7$; the right reports request latency changes in percent. Blue and orange denote MATH500 and MBPP. Error bars are pointwise paired 95% intervals. Stars mark the reference configuration, whose differences are zero; self-comparison intervals are omitted.

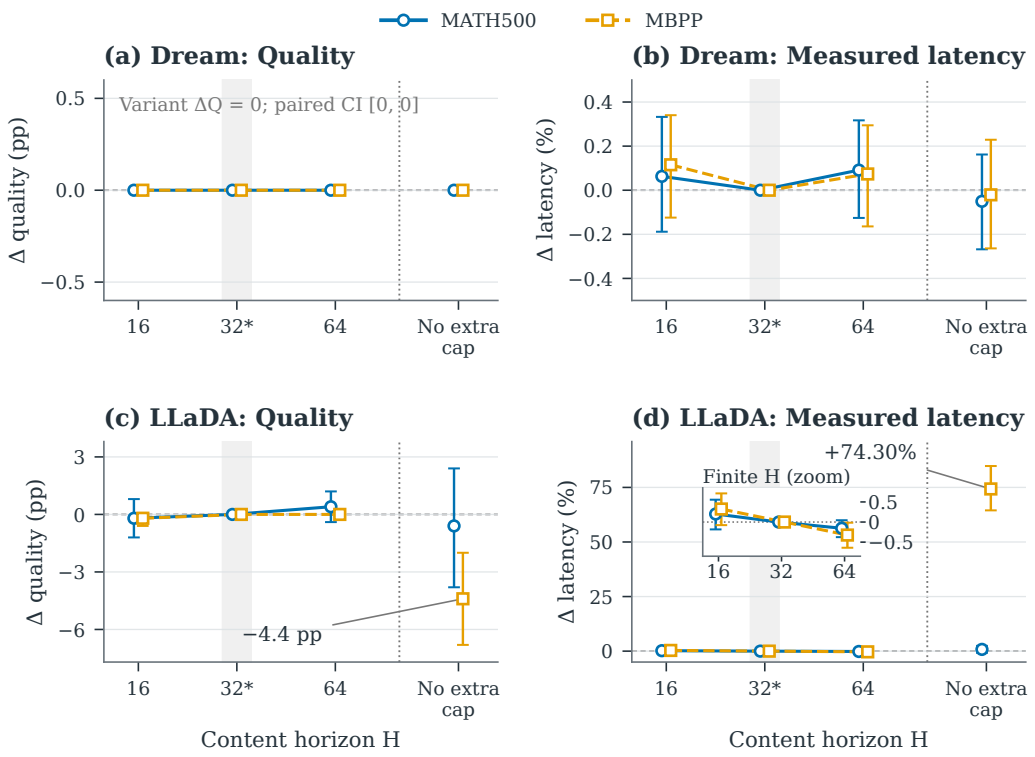
H.2 Content Prefill Horizon

We compare $H \in \{16, 32, 64\}$ and a setting with no additional distance cap. H controls only the extent of LA content prefill written directly to the main state beyond the opened blocks. It does not change the private lane's block length or limit EOS padding. All settings use the same generation canvas and candidate confidence rules.

Quality and latency vary little across the tested finite horizons (Figure 7): all mean latency changes relative to the default are below 0.4% in magnitude. Dream has identical correctness labels for every request across these settings, and LLaDA's task scores vary only slightly. Thus, comparable results are observed at each of the three tested finite horizons on these tasks.

A larger horizon adds positions eligible for screening, but does not guarantee additional useful prefill. For example, extending the horizon from 32 to 64 writes no additional candidates if predictions at the newly eligible positions all fall below the confidence threshold. Dream's activity records likewise show that the vast majority of requests contain no event in which a candidate is excluded by the distance cap. Changing the horizon consequently has few opportunities to alter their prefill choices, helping explain the nearly flat curves.

Removing the distance cap produces a different pattern. On LLaDA MBPP, mean request latency rises by approximately 74%, alongside longer outputs and a lower pass@1 point estimate. The other model–task combinations do not show changes of the same magnitude. LA reuses full-sequence predictions that have already been computed, so expanding the horizon does not itself require an additional model call. It does, however, allow more distant predictions to enter the main state and condition subsequent computation. This can change the generation trajectory and the work needed to complete a request; more direct prefill need not mean faster completion. The concurrent increases in output length and latency provide a clue to this behavior, but do not identify the specific cause.



* Default; pointwise 95% intervals. Each panel uses its own linear y-scale.

Figure 7: **Content prefill horizon sensitivity.** The layout follows Figure 6, with $H = 32$ as the reference. *No extra cap* removes the additional distance limit while retaining the 512-position canvas. The inset in the bottom-right panel enlarges latency changes for the finite horizons $H \in \{16, 32, 64\}$. Panels use independent linear vertical scales. Error bars are pointwise paired 95% intervals; Dream's zero-width quality-difference intervals reflect identical request-level correctness labels in this sample.

These results support retaining a finite content prefill horizon: the tested finite settings give similar results, whereas removing the cap incurs a substantial cost on LLaDA MBPP. They support $H = 32$ as a practical finite horizon on the evaluated tasks.

I Evaluation at 1,024-Token Output Budget

Tables 10 and 11 report full-benchmark results with 1,024 generation slots. The evaluation sets contain GSM8K, MATH500, HumanEval, and MBPP items. All runs use an NVIDIA H100 80GB, bfloat16, and request batch size 1.

Per-run TPS divides total generated tokens before EOS by total request time, excluding model loading, warm-up, and scoring; the tables average these per-run values. Comparison rows report Ours minus the reference in score percentage points, reference/Ours for NFE, and Ours/reference for TPS.

Table 10: **LLaDA-8B-Instruct at 1,024 tokens.** Values are four-seed means; nonzero score SDs are shown after ±. Notation and boldface follow Table 1.

Method	**GSM8K** (5 − *shot*)			**MATH500** (4 − *shot*)			**HumanEval** (0 − *shot*)			**MBPP** (3 − *shot*)		
	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑
Original	79.00	1024.00	2.61	**43.00**	1024.00	8.20	41.46	1024.00	7.33	**38.40**	1024.00	1.63
Fast-dLLM	79.15	110.93	43.95	39.60	217.65	73.75	40.85	130.85	77.36	34.00	79.81	68.67
AdaBlock	80.36	116.23	39.31	41.00	228.04	64.11	**42.07**	141.96	66.48	37.20	79.01	38.28
Ours	**80.43**	**54.12**	**83.63**	40.60	**132.65**	**102.78**	41.54	**79.53**	**111.43**	37.40	**43.30**	**123.13**
vs. Original	(+1.43)	18.92×	31.98×	(−2.40)	7.72×	12.54×	(+0.08)	12.88×	15.19×	(−1.00)	23.65×	75.44×
vs. AdaBlock	(+0.07)	2.15×	2.13×	(−0.40)	1.72×	1.60×	(−0.53)	1.79×	1.68×	(+0.20)	1.82×	3.22×

Table 11: **Dream-v0-Instruct-7B at 1,024 tokens.** Values are four-seed means; nonzero score SDs are shown after ±. Notation and boldface follow Table 1.

Method	**GSM8K** (5 − *shot*)			**MATH500** (4 − *shot*)			**HumanEval** (0 − *shot*)			**MBPP** (3 − *shot*)		
	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑	Score ↑	NFE ↓	TPS ↑
Original	**81.12**	1024.00	1.67	**45.60**	1024.00	4.14	**57.93**	1024.00	3.44	**59.60**	1024.00	0.85
Fast-dLLM	77.86	110.83	29.62	44.00	164.42	55.21	56.10	126.17	43.19	53.80	90.49	17.39
AdaBlock	77.94	111.96	29.17	44.00	164.08	55.11	53.66	125.88	40.04	51.60	93.75	15.36
Ours	79.89 ±0.32	**56.81**	**61.57**	44.60	**88.06**	**98.27**	54.88	**55.20**	**89.96**	56.75 ±0.79	**27.87**	**49.78**
vs. Original	(−1.23)	18.02×	36.95×	(−1.00)	11.63×	23.75×	(−3.05)	18.55×	26.13×	(−2.85)	36.74×	58.62×
vs. AdaBlock	(+1.95)	1.97×	2.11×	(+0.60)	1.86×	1.78×	(+1.22)	2.28×	2.25×	(+5.15)	3.36×	3.24×

J LIMITATIONS

AsyncLane operates within a block-decoding framework. Early advancement reduces waiting at block-completion barriers, but decoding progress and handoff remain organized by blocks. In our LLaDA and Dream implementations, KV cache reuse and refresh also follow this block structure. We have not explored alternative cache organizations beyond this framework. Future work could jointly explore finer-grained KV cache management and decoding schedules, investigating whether computational reuse can be preserved while reducing dependence on block boundaries and full-block completion. Such designs could enable greater decoding parallelism in diffusion language models.