Flash-dLLM: IO-Aware KV Caching and Parallel Decoding for Fast, Memory-Efficient Diffusion LLMs

Quan Nguyen-Tri
VILA Lab, MBZUAI
Abu Dhabi, UAE
quan.nguyen@mbzuai.ac.ae

Mukul Ranjan
VILA Lab, MBZUAI
Abu Dhabi, UAE
mukul.ranjan@mbzuai.ac.ae

Zhiqiang Shen*
VILA Lab, MBZUAI
Abu Dhabi, UAE
zhiqiang.shen@mbzuai.ac.ae

Code available at: https://github.com/VILA-Lab/Flash-dLLM

Abstract

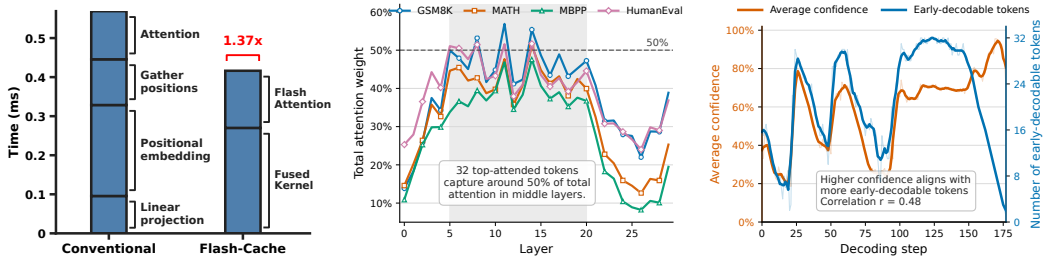
Diffusion Large Language Models (dLLMs) have recently emerged as a promising alternative to autoregressive LLMs by enabling non-autoregressive text generation. However, their practical deployment remains limited by inefficient inference, largely due to the absence of effective Key-Value (KV) caching and scalable parallel decoding mechanisms. Existing acceleration methods typically study KV caching and parallel decoding in isolation, overlooking the I/O bottlenecks that arise when cache reuse and parallel token verification are jointly applied. In this work, we introduce **Flash-dLLM**, a training-free inference acceleration framework for fast and memory-efficient dLLMs. Flash-dLLM first identifies GPU memory I/O as a dominant bottleneck in KV-cache-enabled dLLM inference and addresses it with an I/O-aware fused KV-cache kernel that reduces redundant memory movement. Building on this optimized cache mechanism, Flash-dLLM further proposes an efficient KV-cache-driven draft-and-verify decoding strategy, where the dLLM itself serves as both drafter and verifier without requiring an auxiliary model. This unified design enables faster decoding while preserving generation quality and improving scalability to longer sequences and larger batch size. Extensive experiments on mathematical reasoning and code-generation benchmarks demonstrate that Flash-dLLM consistently outperforms existing state-of-the-art dLLM acceleration methods in both inference speed and memory efficiency. In particular, at generation length 512 it achieves $5.1\times$ and $11.0\times$ higher throughput than Elastic-Cache on GSM8K and HumanEval, respectively.

1 Introduction

Diffusion Large Language Models (dLLMs) [4, 38, 26, 7, 33, 53, 34, 52, 40, 15, 20, 3, 28, 32, 50] have recently emerged as a promising alternative to conventional autoregressive large language models [36, 1, 41, 12, 16, 51] by reformulating text generation as an iterative denoising process. Instead of producing tokens strictly from left to right, dLLMs refine partially masked or noisy sequences over multiple denoising steps, enabling more flexible generation schedules and exposing

*Corresponding author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).



(a) IO bottleneck in KV caching (b) Attention sparsity across layers (c) Confidence and early decoding

Figure 1: Motivating observations (LLaDA-1.5). (a) Per-layer latency breakdown: the conventional cache pipeline is dominated by memory-bound operations; our fused kernel eliminates intermediate materialization for a 1.37× speedup. (b) Top-32 most-attended decoded tokens capture up to 50% of total attention in middle layers, motivating selective tracking over full recomputation. (c) Average prediction confidence and number of early-decodable tokens are positively correlated across denoising steps, indicating that improving confidence directly increases tokens decoded per step.

opportunities for non-autoregressive decoding. This paradigm has shown encouraging potential for controllable generation, global sequence refinement, and improved parallelism. However, despite these algorithmic advantages, the practical inference efficiency of open-source dLLMs still lags behind mature autoregressive LLM systems, whose deployment has benefited from years of optimization around KV caching, attention kernels, and speculative decoding [23, 54, 8, 35].

A major bottleneck to efficient dLLM inference lies in the repeated construction and movement of Key-Value (KV) states across denoising iterations. In autoregressive LLMs, KV caching avoids recomputing attention states for previously generated tokens. In dLLMs, however, the sequence is repeatedly revisited, and the set of updated tokens changes across iterations. This causes frequent KV-cache reads, writes, and updates [27, 42, 30], even when many cached states remain unchanged or contribute little to the current decoding step. As a result, naive KV caching may reduce floating-point computation but introduce substantial GPU memory-access overhead. In practice, the redundancy in KV-cache read and write operations can dominate runtime, limiting the actual speedup obtainable from cache reuse (Fig. 1a).

Beyond this system-level redundancy, dLLM decoding also exhibits strong token-level sparsity. Although each denoising step processes the full sequence, only a small fraction of decoded or partially decoded tokens has a significant influence on the current prediction distribution. Many tokens remain stable across iterations or provide limited additional context for the next decoding decision. Treating all cached tokens as equally important therefore wastes memory bandwidth and computation [42, 19, 57, 49, 43]. This observation suggests that efficient dLLM inference should not simply cache and reuse all KV states uniformly, instead, it should prioritize the subset of tokens that meaningfully affect the current decoding process while reducing unnecessary memory traffic for less influential tokens (Fig. 1b).

Another key point comes from the confidence dynamics of dLLM generation [44, 22, 21, 48]. During iterative denoising, many tokens become semantically determined at an early stage, even though their individual confidence scores may not yet exceed the conservative threshold required for immediate commitment. These tokens are often close to being correctly decoded, but existing decoding strategies delay their acceptance until later iterations, leading to redundant refinement steps. This creates a mismatch between token readiness and token commitment: the model already contains sufficient information to recover many positions, yet the decoding algorithm fails to exploit this early predictability due to insufficient confidence calibration.

Importantly, raising the average confidence of candidate tokens directly increases the number of tokens that can be accepted correctly per denoising step (Fig. 1c). The bottleneck is thus not only whether tokens can be predicted early, but whether their confidence is reliable enough for safe parallel commitment. A better decoding strategy should aggregate useful context, suppress low-impact cache interactions, and raise the confidence of promising candidates, so that dLLMs decode more tokens per iteration without sacrificing quality.

Motivated by these observations, we propose **Flash-dLLM**, a training-free inference framework that jointly optimizes IO-aware KV caching and cache-driven parallel decoding for fast, memory-

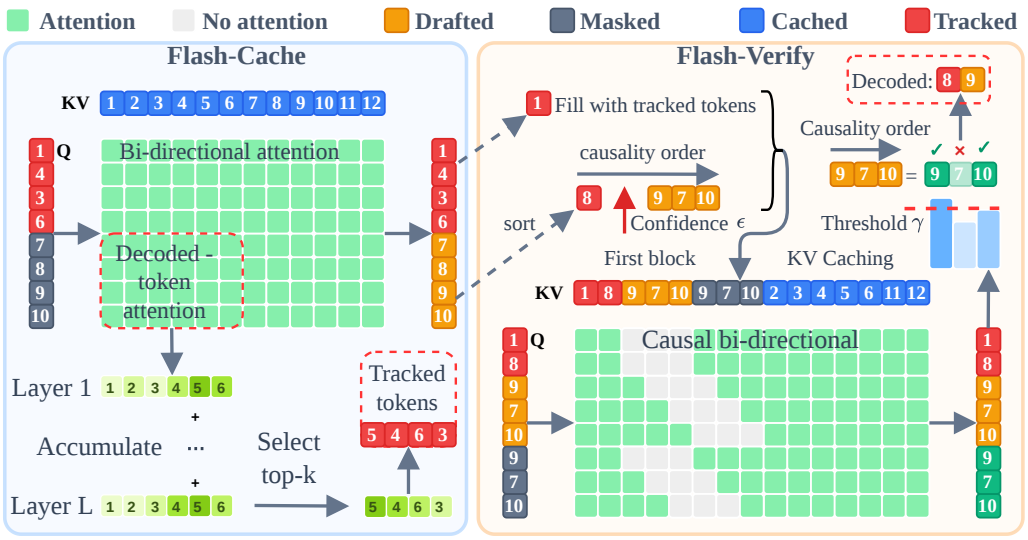


Figure 2: Overview of Flash-dLLM. **Left (Flash-Cache):** At each step, a fixed-size query of β_m tokens (tracked + current masked window) attends bidirectionally to the full KV cache via the fused kernel. Tracked positions are the top-k decoded tokens by attention importance, reselected each step to keep the cache-served representations aligned with full recomputation. **Right (Flash-Verify):** Draft predictions are sorted by confidence. Tokens above threshold ϵ are decoded directly. Remaining search tokens are duplicated in the query (once with the draft prediction, once with `[MASK]`) under a causal attention mask that prevents the two views from attending to each other. Tokens are accepted left-to-right when both views agree and the mask-view confidence exceeds γ.

efficient dLLMs. Flash-dLLM first reduces redundant KV-cache memory movement through an IO-aware fused cache kernel, which minimizes unnecessary read/write operations and improves cache locality during iterative denoising. It then exploits token-level sparsity by focusing cache reuse and verification on tokens that most affect the current decoding step. Finally, Flash-dLLM introduces a KV-cache-based parallel draft-and-verify mechanism that enables the dLLM itself to draft multiple early-decodable tokens and verify them with improved confidence, without requiring an auxiliary drafter or additional training. Together, these designs convert the inherent parallelism of diffusion language generation into practical wall-clock acceleration and improved memory scalability.

This study makes the following contributions:

- We identify redundant read/write memory access as a key bottleneck in KV-cache-enabled dLLM inference, and propose an IO-aware fused KV-cache mechanism to reduce unnecessary GPU memory movement and improve cache locality. We further observe that only a small subset of decoded tokens significantly affects the current decoding step. Based on this, Flash-dLLM selectively prioritizes influential tokens during cache reuse and verification, improving efficiency without uniformly processing all cached states.

- We propose a training-free parallel decoding strategy where the dLLM itself serves as both drafter and verifier. By leveraging cached states to increase candidate-token confidence, Flash-dLLM enables more tokens to be decoded correctly in early stages, reducing denoising steps while preserving generation quality.

- Our approach achieves strong empirical speed and scalability gains. Extensive experiments on mathematical reasoning and code generation benchmarks demonstrate that the proposed Flash-dLLM improves inference speed and memory efficiency while maintaining generation quality, outperforming existing dLLM acceleration methods.

2 Methodology

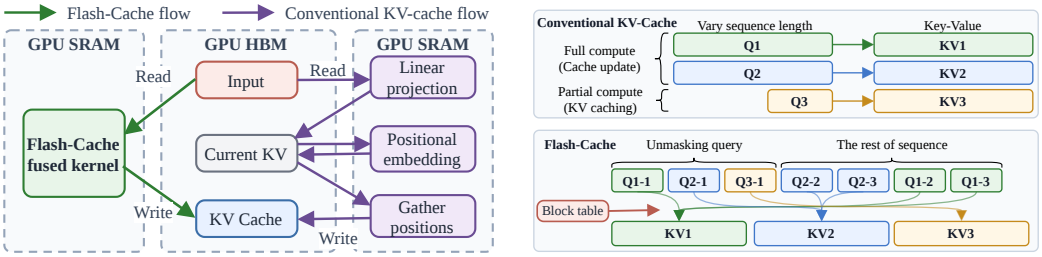
2.1 Preliminaries

KV Caching in Diffusion LLMs. We adopt the sliding window decoding and KV caching strategy from Elastic-Cache [31]. Let $\mathcal{I} = \{1, \ldots, N\}$ represent all positions, $\mathcal{D}^t$ denote the newly decoded positions at step t, and $\mathcal{D}^{<t} = \mathcal{D}^0 \cup \cdots \cup \mathcal{D}^{t-1}$ denote the decoded positions up to step t, and $\mathcal{M}^t$ represent the remaining masked positions. At the initial step, we feed the entire sequence into the model to initialize the KV cache values for all positions $\mathcal{I}$: $\tilde{\mathbf{K}}^{t,l}_{[\mathcal{I}]}$ and $\tilde{\mathbf{V}}^{t,l}_{[\mathcal{I}]}$. For the subsequent step t, we perform KV caching for every token except the current query positions $\mathcal{Q}^t = \mathcal{D}^t \cup \mathcal{M}^t_{\beta_m}$. This includes the newly decoded tokens $\mathcal{D}^t$ and a fixed sliding window of masked tokens $\mathcal{M}^t_{\beta_m} = \mathcal{M}^t_{[1:\beta_m]}$, where β_m is the fixed window size. The attention at step t is computed as follows:

$$\mathbf{A}^{t,l}_{[\mathcal{Q}^t]} = \text{softmax}\left(\frac{\mathbf{Q}^{t,l}_{[\mathcal{Q}^t]}(\tilde{\mathbf{K}}^{t,l}_{[\mathcal{I}]})^\top}{\sqrt{d_k}}\right)\tilde{\mathbf{V}}^{t,l}_{[\mathcal{I}]}, \quad \text{update: } \tilde{\mathbf{K}}^{t,l}_{[\mathcal{Q}^t]} = \mathbf{K}^{t,l}_{[\mathcal{Q}^t]}, \;\; \tilde{\mathbf{V}}^{t,l}_{[\mathcal{Q}^t]} = \mathbf{V}^{t,l}_{[\mathcal{Q}^t]}. \tag{1}$$

The cache is exact for positions in $\mathcal{Q}^t$ and approximate for the rest. Existing methods differ in how they manage this approximation: Fast-dLLM [46] refreshes the entire cache at block boundaries, and Elastic-Cache [31] triggers refresh adaptively based on attention-pattern drift. All these methods implement the cache update logic in PyTorch, issuing separate kernel launches for QKV projection, rotary positional embedding, cache writes, and attention computation per layer. This makes the caching operation memory-bound: the intermediate tensors are written to and read from GPU HBM multiple times, and the memory access cost dominates over the arithmetic cost.

Parallel Decoding in Diffusion LLMs. Diffusion LLMs generate by iteratively unmasking tokens from a fully masked sequence. At each step, the model predicts all masked positions in parallel, but these predictions are conditionally independent given $\mathbf{x}_t$. The model samples from the product of marginals $\prod_i p(\mathbf{x}^i_s \mid \mathbf{x}_t)$, while the true joint $p(\mathbf{x}^i_s, \mathbf{x}^j_s \mid \mathbf{x}_t) = p(\mathbf{x}^i_s \mid \mathbf{x}_t) \cdot p(\mathbf{x}^j_s \mid \mathbf{x}_t, \mathbf{x}^i_s)$ contains inter-token dependencies [46]. Decoding many tokens at once amplifies this discrepancy and degrades coherence. Fast-dLLM [46] tackles this issue by introducing *Confidence-aware* decoding. This approach selectively unmasks only tokens whose confidence $c^i = \max_x p_\theta(x^i \mid \mathbf{x}_t)$ surpasses a predefined threshold ϵ. Consequently, parallel decoding effectively approximates the true joint distribution when all decoded tokens exhibit high confidence.



(a) Key-Value (KV) Caching in Flash-Cache (b) Flash Attention in Flash-Cache

Figure 3: IO-aware memory management in Flash-Cache. (a) The fused kernel eliminates intermediate KV tensors by performing projection, RoPE, and cache write directly on the KV cache, reducing HBM read/write traffic and memory usage. (b) Varying sequence lengths from partial (cached) and full (updated) computations are reorganized into contiguous blocks managed by a block table, enabling parallel execution without padding or synchronization overhead.

2.2 Flash-Cache: IO-aware key-value caching

Flash-Cache consists of three main components: (i) a fused kernel for Key-Value (KV) caching to alleviate the IO bottleneck, (ii) scheduled flash attention to manage the substantial variation in sequence lengths within batches, and (iii) a selective cache update that tracks only the most-attended tokens.

Algorithm 1 Flash-Cache

Require: Input $\mathbf{X} \in \mathbb{R}^{M \times d}$, query $\mathbf{Q} \in \mathbb{R}^{M \times d}$, KV cache $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{(BN) \times d}$, output $\mathbf{A} \in \mathbb{R}^{M \times d}$, positions $\mathbf{P} \in \mathbb{R}^{M \times d}$, block size β, sequence length N, query length M, batch size B, heads h

Split $\mathbf{X}$ into $T_m = M/\beta$ blocks $\{\mathbf{X}_i\}_{i=1}^{T_m}$; create block table $\mathbf{B}$ and assign batch index $(\mathbf{X}_i, b)$
Split each block into heads $\mathbf{X}_{i,1}, \ldots, \mathbf{X}_{i,h}$

Flash Fused Kernel

for $1 \le i \le T_m$ **do**
  Load $\mathbf{p} \leftarrow \mathbf{P}_i$ *// HBM → SRAM*
  for $1 \le j \le h$ **do**
    Load $\mathbf{x} \leftarrow \mathbf{X}_{i,j}$ *// HBM → SRAM*
    $\mathbf{q}, \mathbf{k}, \mathbf{v} \leftarrow \texttt{projection}(\mathbf{x})$; $\mathbf{q}, \mathbf{k} \leftarrow \texttt{RoPE}(\mathbf{q}, \mathbf{k})$
    Write $\mathbf{q} \rightarrow \mathbf{Q}_{i,j}$
    Write $\mathbf{k}, \mathbf{v} \rightarrow \mathbf{K}_{[\mathbf{p},j]}, \mathbf{V}_{[\mathbf{p},j]}$ *// SRAM → HBM*
  end for
end for

Scheduled Flash Attention

for $1 \le i \le T_m$ **do**
  Load $b \leftarrow \mathbf{B}_i$ *// HBM → SRAM*
  for $1 \le j \le h$ **do**
    Load $\mathbf{q} \leftarrow \mathbf{Q}_{i,j}$ *// HBM → SRAM*
    Load $\mathbf{k}, \mathbf{v} \leftarrow \mathbf{K}_{[b,j]}, \mathbf{V}_{[b,j]}$ *// HBM → SRAM*
    $\mathbf{a} \leftarrow \texttt{attention}(\mathbf{q}, \mathbf{k}, \mathbf{v})$
    Write $\mathbf{a} \rightarrow \mathbf{A}_{i,j}$ *// SRAM → HBM*
  end for
end for

KV caching fused kernel. At each transformer layer, a conventional KV cache implementation launches four separate CUDA kernels for the query set $\mathcal{Q}^t$: QKV projection, rotary positional embedding (RoPE), cache write, and attention. Each kernel writes its output to GPU high-bandwidth memory (HBM) before the next kernel reads it. QKV projection, RoPE, and cache writing each incur $\mathcal{O}(Qd_{\text{model}})$ memory traffic, while attention streams over the full cache with $\mathcal{O}(Nd)$ traffic. Thus, the total per-layer HBM traffic is approximately $4 \times \mathcal{O}(Qd_{\text{model}}) + \mathcal{O}(Nd)$. Because the non-attention operations have low arithmetic intensity, the cache-update path becomes memory-bound (Fig. 3a).

Inspired by Flash Attention [13], we introduce a fused Flash-Cache kernel that combines QKV projection, RoPE, and cache writing (Fig. 3a). The fused kernel performs projection and RoPE in SRAM and writes the resulting keys and values directly to the KV cache, eliminating intermediate key-value materialization. This reduces HBM traffic, lowers memory usage, and improves IO efficiency. As shown in Fig. 1a, positional embedding and cache writing can dominate runtime relative to core computations such as attention and QKV projection. With Flash-Cache, we achieve a $1.37\times$ speedup on an RTX 3090 GPU.

Scheduled flash attention. KV caching and parallel decoding introduce new challenges for scaling diffusion LLMs to batched inference. While conventional attention can process variable-length sequences through padding and length-based grouping, diffusion LLMs make sequence lengths more dynamic and divergent. KV caching typically alternates between a caching stage, which computes only over a small window, and an update stage, which recomputes the full sequence; as a result, samples in the same batch may require substantially different computation at each iteration. Forcing all samples into the same stage can reduce efficiency and accuracy. This issue is further amplified by adaptive methods such as Elastic-Cache, where sequence length may vary within a layer, and by parallel decoding, which causes samples to progress at different rates.

To address this challenge, we extend Flash Attention [13, 39, 56] by partitioning each batch into multiple sequence blocks and scheduling them with a block table that aligns query blocks with their corresponding key-value blocks (Fig. 3b). This design mitigates length discrepancies during inference and provides a flexible mechanism for adding or removing blocks, enabling more adaptive decisions about when to cache or update. Unlike Flash Attention, which primarily optimizes IO efficiency within the attention computation, our Scheduled Flash Attention focuses on controlling which blocks are computed and in what order through the block table. This scheduling strategy is particularly effective for accelerating batched inference in diffusion LLMs.

The overall algorithm of Fused Kernel and Scheduled Flash Attention is presented in algorithm 1.

Selective cache update. Building on the fused kernel and scheduled flash attention, we further improve scalability with a simple Selective cache update. We observe that the 32 most-attended decoded tokens account for up to 50% of the attention weight in the middle layers (layers 5–20, Fig. 1b). This suggests that refreshing only a small subset of tokens preserves most of the information that a full cache refresh would provide. Motivated by this observation, we maintain and update only a fixed set of the most-attended tokens (Algorithm 2).

Our method constructs a fixed-size query at each subsequent decoding step ($t > 0$) using a sliding unmasking window of size β_m and a fixed tracking budget β_t. Specifically, the query for step $t + 1$

Algorithm 2 Flash-dLLM

Require: Model f_θ, prompt $\mathbf{x}_{\text{prompt}}$, generation length N, tracking budget β_t, masked-window size β_m, confidence threshold ϵ, verify threshold γ.

$\mathbf{x}^0 \leftarrow \{\mathbf{x}_{\text{prompt}};\ \texttt{[MASK]}^N\}$; $\mathcal{D}^{<1} \leftarrow \{1, \ldots, |\mathbf{x}_{\text{prompt}}|\}$; $\mathcal{M}^1 \leftarrow \{|\mathbf{x}_{\text{prompt}}|+1, \ldots, |\mathbf{x}_{\text{prompt}}|+N\}$;
Allocate KV cache $\tilde{\mathbf{K}}^l, \tilde{\mathbf{V}}^l \in \mathbb{R}^{(B \cdot N_{\max}) \times d_{\text{model}}}$ for each layer l *// pre-allocate once*
while $\mathcal{M}^t \neq \emptyset$ **do**
  $\mathcal{M}^t_{\beta_m} \leftarrow \mathcal{M}^t_{[:\beta_m]}$; $\mathcal{Q}^t \leftarrow \mathcal{T}^t \cup \mathcal{M}^t_{\beta_m}$
  $p_\theta(\mathbf{x} \mid \mathbf{x}^t)$, $\mathbf{S} \leftarrow \texttt{FusedForward}(f_\theta, \mathbf{x}^t_{[\mathcal{Q}^t]}, \tilde{\mathbf{K}}, \tilde{\mathbf{V}})$ *// Flash-Cache, Alg. 1*
  $c^i, \hat{x}^i \leftarrow \max_x p_\theta(x^i \mid \mathbf{x}^t)$ for $i \in \mathcal{M}^t_{\beta_m}$; $\mathcal{D}^t \leftarrow \{i \in \mathcal{M}^t_{\beta_m} : c^i \geq \epsilon\}$; $\mathcal{S}^t \leftarrow \mathcal{M}^t_{\beta_m} \setminus \mathcal{D}^t$
  if `verify` **and** $\mathcal{S}^t \neq \emptyset$ **then** *// Flash-Verify*
    Adjust tracking set $\mathcal{T}^t_v \leftarrow \mathcal{T}^t_{[:2\beta_m - |\mathcal{D}^t| - 2|\mathcal{S}^t|]}$
    Construct verify query: $\{\mathbf{x}^t_{[\mathcal{T}^t_v]}; \hat{\mathbf{x}}_{[\mathcal{D}^t]};\ \hat{\mathbf{x}}_{[\mathcal{S}^t]}; \mathbf{x}^t_{[\mathcal{S}^t]}\}$
    Construct causal mask for bi-directional attention $\mathbf{M}$
    $p_\theta(\mathbf{x} \mid \hat{\mathbf{x}}_{[\mathcal{S}^t]}) \leftarrow \texttt{FusedForward}(f_\theta, \text{verify query}, \tilde{\mathbf{K}}, \tilde{\mathbf{V}}, \mathbf{M})$
    $\tilde{c}^i, \tilde{x}^i \leftarrow \max_x p_\theta(x^i \mid \hat{\mathbf{x}}_{[\mathcal{S}^t]})$ for $i \in \mathcal{S}^t$
    $\mathcal{D}^t \leftarrow \mathcal{D}^t \cup \{i \in \mathcal{S}^t : \hat{x}^i = \tilde{x}^i \ \wedge\ \tilde{c}^i \geq \gamma, \text{ up to first mismatch}\}$
  end if
  Decode: $\mathbf{x}^{t+1}_{[\mathcal{D}^t]} \leftarrow \hat{\mathbf{x}}_{[\mathcal{D}^t]}$; $\mathcal{D}^{<t+1} \leftarrow \mathcal{D}^{<t} \cup \mathcal{D}^t$; $\mathcal{M}^{t+1} \leftarrow \mathcal{M}^t \setminus \mathcal{D}^t$
  $a^t_i \leftarrow \sum_l \frac{1}{H|\mathcal{M}^t_{\beta_m}|} \sum_{h,\, j \in \mathcal{M}^t_{\beta_m}} \mathbf{S}^{t,l,h}_{j,i}$ for $i \in \mathcal{D}^{<t}$
  $\mathcal{T}^{t+1} = \text{top-k}(\{a^t_i : i \in \mathcal{D}^{<t}\},\ \beta_t - |\mathcal{D}^t|) \cup \mathcal{D}^t$ $t \leftarrow t+1$
end while
return $\mathbf{x}^t$

is defined as $\mathcal{Q}^{t+1} = \mathcal{M}^{t+1}_{\beta_m} \cup \mathcal{T}^{t+1}$, where $\mathcal{T}^{t+1}$ comprises the newly decoded tokens $\mathcal{D}^t$ and the most-attended tokens from the preceding step. The latter are selected from previously decoded tokens according to the attention they receive from masked tokens. The attention score of each decoded token i at step t is computed as follows:

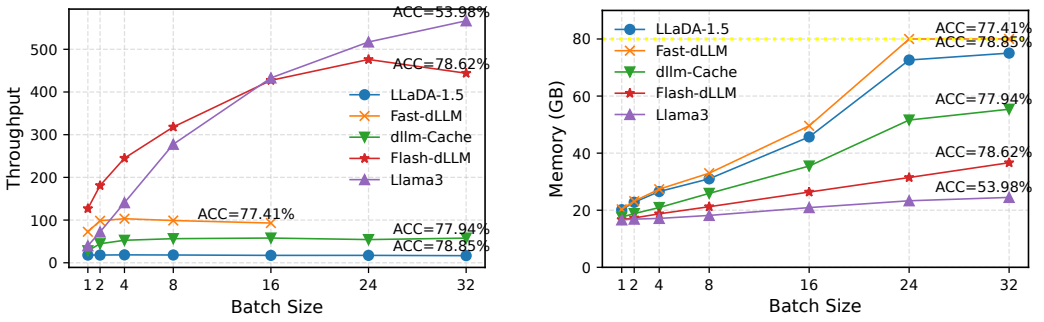
$$a^t_i = \sum_{l=1}^{L} \frac{1}{H} \sum_{h=1}^{H} \frac{1}{|\mathcal{M}^t_{\beta_m}|} \sum_{j \in \mathcal{M}^t_{\beta_m}} \mathbf{S}^{t,l,h}_{j,i}, \quad i \in \mathcal{D}^{<t} \tag{2}$$

where $\mathbf{S}^{t,l}$ is unnormalized attention logits. The score measures the extent to which the current masked queries are paying attention to the decoded position i. The tracking set is $\mathcal{T}^{t+1} = \text{top-k}(\{a^t_i : i \in \mathcal{D}^{<t}\},\ \beta_t - |\mathcal{D}^t|) \cup \mathcal{D}^t$. Newly decoded tokens are prepended to $\mathcal{D}^{<t}$ before ranking, giving them automatic inclusion. All other decoded positions are served from cache without participating as queries, bounding per-step compute at $\beta_t + \beta_m$.

2.3 Flash-Verify: KV-cache-driven draft-and-verify Parallel Decoding

Confidence-aware decoding [46, 45] unmasks only tokens whose confidence $c^i = \max_x p_\theta(x^i \mid \mathbf{x}_t)$ exceeds a threshold ϵ, discarding all others even when many are correct (Fig. 1c). This creates a throughput ceiling on tasks where the model is uncertain, as few tokens pass the threshold per step. We propose Flash-Verify, a self-verification scheme in which the dLLM serves as both drafter and verifier, recovering correct predictions that confidence-aware decoding would waste. At each denoising step, a standard **draft pass** runs the model on $\mathcal{Q}^t = \mathcal{T}^t \cup \mathcal{M}^t_{\beta_m}$ against the full KV cache. The masked positions are sorted by confidence and partitioned into a confident set $\mathcal{D}^t$ (above ϵ, accepted directly) and a search set $\mathcal{S}^t$ (below ϵ, candidates for verification).

A **verify pass** then constructs a new query with three groups: an adjusted tracking set $\mathcal{T}_v$ consisting of previously decoded tokens and $\mathcal{D}^t$, the search positions $\mathcal{S}^t$ filled with their draft predictions $\hat{x}^i$ (the *draft view*), and the same positions filled with `[MASK]` (the *mask view*). Both views share positional embeddings but are isolated by a causal attention mask loaded inside the fused Triton kernel: the tracked context cannot attend to the draft view, and the draft and mask views at the same position cannot attend to each other, so they produce independent predictions from shared context. A



(a) Throughput vs. batch size

(b) Peak memory vs. batch size

Figure 4: Scalability on GSM8K-512 (1-shot, LLaDA-1.5). (a) Flash-dLLM (Flash-Cache+Flash-Verify) throughput rises with batch size and saturates around batch size 24–32, while Fast-dLLM stays nearly flat and runs out of memory at 24. Llama3-8B is an autoregressive reference. (b) Flash-dLLM uses less memory than all diffusion-LLM baselines across batch sizes.

search token is accepted if both views agree and the mask-view confidence exceeds a threshold γ: $\text{accept}(i) = \mathbb{I}[\hat{x}^i = \tilde{x}^i] \cdot \mathbb{I}[\tilde{c}^i \geq \gamma]$,

where $\tilde{x}^i$ and $\tilde{c}^i$ are the mask view's prediction and confidence. Following speculative decoding conventions [24], tokens are accepted sequentially along the decoding order and all tokens after the first mismatch are rejected. Appendix C shows that each accepted token deviates from the model's own sequential conditional by at most $1 - \gamma$ in total variation, and a committed block of m tokens by at most $m(1 - \gamma)$.

The verify pass reuses the same fused kernel and pre-allocated KV cache; only the query tokens and the attention mask change, so the additional cost is proportional to $2\beta_m$ rather than the full sequence. Unlike prior draft-and-verify methods for dLLMs that rely on a separate autoregressive verifier [18] or multiple independent forward passes [47], Flash-Verify requires no external model: the dLLM verifies its own predictions through the two-view attention mask, roughly doubling the tokens accepted per step (Fig. 6). Algorithm 2 summarizes the full procedure of Flash-dLLM, including both Flash-Cache and Flash-Verify.

3 Experiments

3.1 Experimental Setup

Implementation Details. All experiments run on a single NVIDIA A100 80GB GPU. We evaluate **Flash-dLLM** on LLaDA-1.5 [59] across GSM8K [11], MATH [17], HumanEval [10], and MBPP [5]. We implement the fused KV-cache kernel in Triton 2.0. Default benchmark is GSM8K, with default hyperparameters: confidence threshold $\epsilon = 0.9$, verify threshold $\gamma = 0.8$, block size $\beta = 16$, tracked budget $\beta_t = 80$, sliding window size $\beta_m = 64$, generation length 512. For fair comparison, we re-run all baselines under identical hardware and software configurations. **Baselines.** We compare against three approaches: (1) **No Cache**: standard dLLM inference without KV caching, under both greedy (fixed-step) and confidence-aware decoding; (2) **Fast-dLLM** [46]: block-wise KV caching with confidence-aware decoding; (3) **Elastic-Cache** [31]: adaptive KV caching with attention-pattern-based cache reuse. We report Flash-dLLM results under three configurations: greedy decoding (pure KV-cache speedup), confidence-aware decoding (KV-cache + parallel decoding), and Flash-Verify (KV-cache + draft-and-verify parallel decoding).

3.2 Main Results

Throughput. Table 1 compares accuracy and decoding efficiency across benchmarks. Flash-Cache substantially accelerates greedy decoding, yielding speedups of $8.0\times$–$58.5\times$. Under confidence-aware decoding, the range increases to $17.0\times$–$102.2\times$, indicating that cache acceleration remains effective with parallel decoding. Combining Flash-Verify and Flash-Cache achieves the highest throughput in all eight settings, reaching 148.0–210.6 tokens/s and speedups of $22.3\times$–$148.2\times$. Relative to confidence-aware Flash-Cache, the second-fastest configuration throughout, it improves

Table 1: Accuracy and decoding efficiency of LLaDA-1.5 across different benchmarks and decoding configurations. Each cell reports accuracy (top) and throughput with speedup over greedy decoding without caching (bottom; blue: tokens/s, orange: speedup). **Bold** indicates the highest accuracy in each row, while yellow shading indicates the highest throughput.

		Greedy		Confidence-Aware				Flash-Verify	
Benchmark	**Len**	**No Cache**	**Flash-Cache**	**No Cache**	**Fast-dLLM**	**Elastic-Cache**	**Flash-Cache**	**No Cache**	**Flash-Cache**
GSM8K (5-shot)	256	80.36 6.7 (1.0×)	82.87 56.8 (8.5×)	80.44 22.5 (3.4×)	80.59 51.2 (7.6×)	81.88 45.9 (6.9×)	82.34 144.9 (21.6×)	**83.62** 38.2 (5.7×)	81.88 194.9 (29.1×)
	512	81.35 2.6 (1.0×)	82.94 54.8 (21.1×)	81.88 17.2 (6.6×)	80.82 36.8 (14.2×)	82.79 41.7 (16.0×)	82.87 149.4 (57.5×)	82.71 32.2 (12.4×)	**83.02** 210.6 (81.0×)
MATH (4-shot)	256	33.52 8.5 (1.0×)	**37.22** 67.6 (8.0×)	33.60 22.3 (2.6×)	32.74 44.4 (5.2×)	33.26 40.6 (4.8×)	36.80 144.3 (17.0×)	36.98 38.1 (4.5×)	36.56 189.7 (22.3×)
	512	35.63 5.0 (1.0×)	37.40 66.2 (13.2×)	35.56 20.3 (4.1×)	33.68 44.4 (8.9×)	35.84 41.4 (8.3×)	37.08 149.9 (30.0×)	**37.76** 32.8 (6.6×)	35.98 210.1 (42.0×)
HumanEval (0-shot)	256	**43.29** 7.0 (1.0×)	42.68 79.0 (11.3×)	42.68 17.5 (2.5×)	34.75 18.7 (2.7×)	36.59 20.9 (3.0×)	40.85 169.4 (24.2×)	37.20 63.2 (9.0×)	39.63 209.2 (29.9×)
	512	40.85 3.2 (1.0×)	41.46 74.5 (23.3×)	39.63 9.7 (3.0×)	36.59 15.4 (4.8×)	37.80 16.8 (5.2×)	**42.07** 145.7 (45.5×)	37.20 57.5 (18.0×)	40.24 185.6 (58.0×)
MBPP (3-shot)	256	38.00 2.4 (1.0×)	41.20 62.0 (25.8×)	38.00 14.2 (5.9×)	34.60 28.0 (11.7×)	41.20 32.7 (13.6×)	**41.80** 115.9 (48.3×)	41.40 37.7 (15.7×)	38.20 148.0 (61.7×)
	512	38.20 1.0 (1.0×)	39.80 58.5 (58.5×)	38.60 11.5 (11.5×)	36.20 17.8 (17.8×)	39.00 32.8 (32.8×)	**40.20** 102.2 (102.2×)	39.40 31.6 (31.6×)	39.00 148.2 (148.2×)

Table 2: Comparison with additional baselines on GSM8K (5-shot, generation length 512, LLaDA-1.5). Each method reports accuracy (%) and throughput in tokens/s, with the speedup over greedy decoding without caching in parentheses; **bold** marks the best accuracy.

Metric	**dKV-Cache**	**FlashDLM**	**dLLM-Cache**	**Dyna-dLLM**	**Fast-dLLM**	**FreeDave**	**Elastic-Cache**	**Flash-Cache (conf-aware)**	**Flash-Cache + Flash-Verify**
Acc. (%)	81.50	79.91	80.97	79.32	80.82	80.97	82.79	82.87	**83.02**
TPS	14.9 (5.7×)	15.7 (6.0×)	16.8 (6.5×)	38.4 (14.8×)	36.8 (14.2×)	42.8 (16.5×)	41.7 (16.0×)	149.4 (57.5×)	210.6 (81.0×)

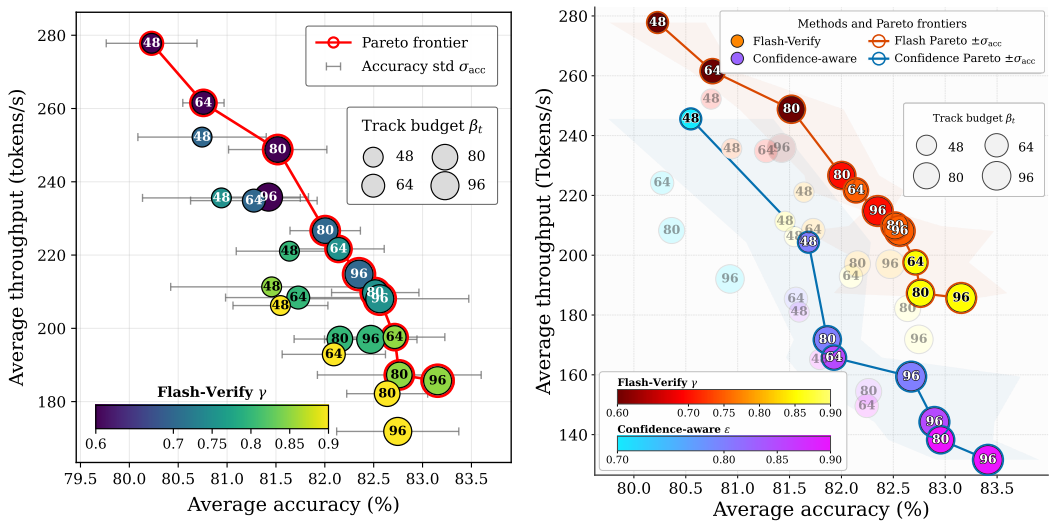
throughput by approximately 23.5%–45.0%. The gains are larger at longer generation lengths: from 256 to 512 tokens, the speedup increases from 29.1× to 81.0× on GSM8K, 22.3× to 42.0× on MATH, 29.9× to 58.0× on HumanEval, and 61.7× to 148.2× on MBPP.

Accuracy. Accuracy exhibits a task-dependent trade-off. On GSM8K-512, Flash-Verify with Flash-Cache achieves both the highest accuracy (83.02%) and throughput (210.6 tokens/s). Elsewhere, the fastest configuration is not consistently the most accurate: Flash-Verify without caching performs best on GSM8K-256 and MATH-512, whereas confidence-aware Flash-Cache leads on HumanEval-512 and both MBPP settings. On mathematical reasoning tasks, the combined method remains within 1.78 percentage points of the best accuracy. Larger gaps arise for 256-token code generation, reaching 3.66 points on HumanEval and 3.60 points on MBPP. Overall, Flash-Verify with Flash-Cache provides the strongest throughput-oriented configuration.

More baselines. Table 2 compares the accuracy and decoding throughput of the evaluated cache and parallel-verification methods. Conventional baselines achieve accuracies of 79.32%–82.79% and throughputs of 14.9–42.8 tokens/s. Elastic-Cache yields the highest baseline accuracy (82.79%), whereas FreeDave achieves the highest baseline throughput (42.8 tokens/s). Flash-Cache with confidence-aware decoding achieves 82.87% accuracy and 149.4 tokens/s, while its integration with Flash-Verify further improves performance to 83.02% accuracy and 210.6 tokens/s.

3.3 Ablation Studies and Analysis

Scalability. Figure 4a reports throughput as a function of batch size on GSM8K-512 (1-shot) using LLaDA-1.5. Flash-dLLM's throughput rises with batch size and saturates around batch size 24–32 without exhausting GPU memory, whereas Fast-dLLM's throughput stays nearly flat and it runs out of memory at batch size 24. Table 6 (Appendix G) extends this analysis to GSM8K-512 (5-shot), showing that all Flash-dLLM configurations (greedy, confident, verify) scale monotonically from batch sizes 1 to 32. In particular, Flash-Cache with Flash-Verify reaches 199.8 tokens/s at batch size 32. Figure 4b further shows that, at batch size 16, Flash-dLLM uses approximately 26 GB of GPU memory, compared with 50 GB for Fast-dLLM, representing a reduction of about 48%. This improvement arises from the flat, preallocated cache layout, which avoids dynamic memory allocation and the padding overhead associated with conventional four-dimensional KV-cache implementations.



(a) Pareto frontier of Flash-Verify (b) Pareto frontiers of Flash-Verify and Confidence

Figure 5: Accuracy–throughput trade-off under different track-budget values β_t and denoising settings γ/ϵ. The mean and standard deviation are computed over five random seeds.

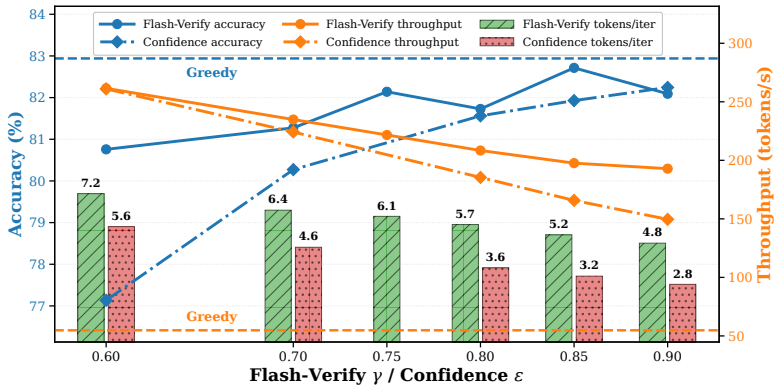


Figure 6: Tokens-per-iteration comparison of Flash-Verify and Confidence-aware decoding.

Accuracy–throughput trade-off. Figure 5a characterizes the accuracy–throughput trade-off across track budgets β_t and Flash-Verify thresholds γ. In general, accuracy and throughput are inversely related: increasing accuracy from approximately 80.2% to 83.2% reduces throughput from 278 to 186 tokens/s. Larger track budgets generally improve accuracy at the cost of additional computation, whereas γ provides finer control over parallelism within a fixed budget. In particular, a favorable trade-off can be obtained by maintaining a relatively large β_t while reducing γ to promote greater decoding parallelism.

Flash-Verify vs. confidence-aware decoding. Figure 5b compares Flash-Verify with confidence-aware decoding. Across most of their shared accuracy range, Flash-Verify delivers substantially higher throughput. At approximately 82.6%–82.9% accuracy, it achieves 190–210 tokens/s, compared with 140–160 tokens/s for confidence-aware decoding. Although confidence-aware decoding attains a slightly higher peak accuracy of approximately 83.4%, its throughput decreases to about 131 tokens/s. Overall, Flash-Verify provides a more favorable accuracy–throughput trade-off.

Tokens decoded per iteration. Figure 6 compares accuracy, throughput, and tokens decoded per iteration across Flash-Verify thresholds γ and confidence-aware thresholds ϵ. Across denoising settings, Flash-Verify generally decodes more tokens per iteration and achieves higher throughput, demonstrating greater parallel-decoding efficiency. Because Flash-Verify incurs an additional verification step, decoding 7.2 tokens per iteration yields throughput comparable to confidence-aware decoding at 5.6 tokens per iteration; however, Flash-Verify achieves 3.5 percentage points higher

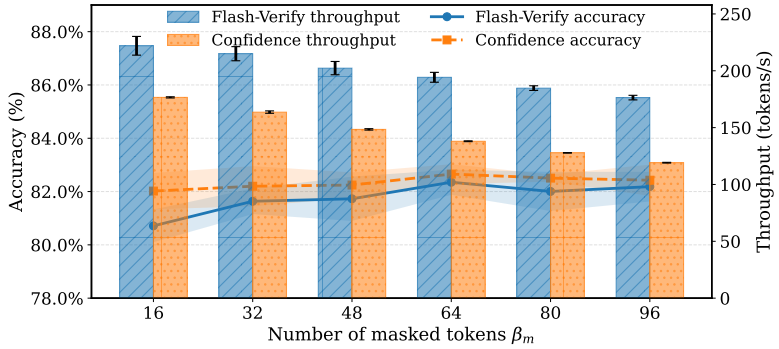


Figure 7: Impact of the masked-window size β_m on accuracy and throughput

accuracy. As the number of tokens decoded per iteration increases, the accuracy gap between the methods narrows, while Flash-Verify's throughput advantage grows to as much as $1.33\times$.

Masked-window size β_m. Figure 7 examines the effect of the masked-window size β_m on the accuracy and throughput of Flash-Verify and confidence-aware decoding. For both methods, accuracy improves up to an intermediate window size, whereas throughput decreases as β_m increases. Flash-Verify achieves comparable accuracy across all settings, with gaps of only 0.2–1.2 percentage points. Moreover, it consistently delivers higher throughput, with its speedup over confidence-aware decoding increasing from approximately $1.4\times$ to $1.5\times$ as the window size grows.

4 Related Work

Diffusion Language Models and Acceleration. Masked diffusion models generate text by iteratively unmasking tokens predicted in parallel [26, 4, 38, 40, 58]. Scaling this paradigm has produced models competitive with autoregressive LLMs, including LLaDA [33], Dream [53], and Gemini Diffusion [15]. However, the lack of KV caching support makes dLLM inference slow: bidirectional attention and evolving hidden states across denoising steps prevent direct reuse of autoregressive caching strategies [35]. Fast-dLLM [46] introduces block-wise approximate KV caching together with confidence-aware decoding. dKV-Cache [29] caches at fixed temporal intervals. Elastic-Cache [31] makes updates adaptive via attention-pattern drift detection. FlashDLM [18] combines token-level change detection with an external autoregressive verifier.

Parallel Decoding and Speculative Verification. Speculative decoding [24, 9] accelerates autoregressive LLMs by drafting tokens with a fast model and verifying them in parallel with the target model. In diffusion LLMs, parallel decoding takes a different form: confidence-aware methods [46] unmask all tokens above a threshold per step, Prophet [25] dynamically halts refinement when predictions stabilize, and FreeDave [47] drafts tokens in one forward pass then verifies in a second. FlashDLM [18] uses an external autoregressive LLM as the verifier. Our Flash-Verify differs from all of these: the dLLM serves as both drafter and verifier through the KV cache, with no external model and no training. Tokens are accepted only when the draft and mask views agree, combining the throughput of aggressive parallel decoding with the accuracy of conservative single-token unmasking.

5 Conclusion

We proposed **Flash-dLLM**, a training-free inference acceleration framework for diffusion LLMs that jointly addresses IO-aware KV caching and parallel draft-and-verify decoding. By reducing redundant KV-cache reads and writes, prioritizing influential decoded tokens, and raising the confidence of early token commitments, Flash-dLLM turns the inherent parallelism of dLLMs into practical speedup. Experiments on mathematical reasoning and code generation show higher speed, lower memory use, and better scalability than existing acceleration methods, offering a practical path toward efficient deployment of diffusion language models.

Acknowledgments and Disclosure of Funding

This work is supported by the MBZUAI-WIS Joint Program for Artificial Intelligence Research.

Competing interests. The authors declare no competing interests.

References

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

[2] Sudhanshu Agrawal, Risheek Garrepalli, Raghavv Goel, Christopher Lott, Fatih Porikli, and Mingu Lee. Structuring the future: Diffusion llm speculative decoding via calibrated draft graphs. *arXiv preprint arXiv:2509.18085*, 2025.

[3] Marianne Arriola, Aaron Gokaslan, Justin Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. In *International Conference on Learning Representations*, volume 2025, pages 50726–50753, 2025.

[4] Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. Structured denoising diffusion models in discrete state-spaces. *Advances in Neural Information Processing Systems*, 34:17981–17993, 2021.

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

[6] Heli Ben-Hamu, Itai Gat, Daniel Severo, Niklas S Nolte, and Brian Karrer. Accelerated sampling from masked diffusion models via entropy bounded unmasking. *Advances in Neural Information Processing Systems*, 38:55981–56007, 2025.

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

[8] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

[9] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

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

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

[12] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

[13] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[14] Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, 07 2024.

[15] Google DeepMind. Gemini diffusion. `https://deepmind.google/models/gemini-diffusion`, 2025. Accessed: 2026-05-02.

[16] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[17] 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. *arXiv preprint arXiv:2103.03874*, 2021.

[18] Zhanqiu Hu, Jian Meng, Yash Akhauri, Mohamed S Abdelfattah, Jae-sun Seo, Zhiru Zhang, and Udit Gupta. Flashdlm: Accelerating diffusion language model inference via efficient kv caching and guided diffusion. *arXiv preprint arXiv:2505.21467*, 2025.

[19] Jianuo Huang, Yaojie Zhang, Yicun Yang, Benhao Huang, and Linfeng Zhang. Mask tokens as prophet: Fine-grained cache eviction for efficient dllm inference. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 3456–3479, 2026.

[20] Inception Labs. Introducing mercury: The first commercial diffusion-based language model. `https://www.inceptionlabs.ai/introducing-mercury`, 2025. Accessed: 2026-05-02.

[21] Daniel Israel, Guy Van den Broeck, and Aditya Grover. Accelerating diffusion llms via adaptive parallel decoding. *Advances in neural information processing systems*, 38:52870–52888, 2025.

[22] Fanheng Kong, Jingyuan Zhang, Yahui Liu, Zirui Wu, Yu Tian, Guorui Zhou, et al. Accelerating diffusion llm inference via local determinism propagation. *arXiv preprint arXiv:2510.07081*, 2025.

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

[24] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[25] Pengxiang Li, Yefan Zhou, Dilxat Muhtar, Lu Yin, Shilin Yan, Li Shen, Soroush Vosoughi, and Shiwei Liu. Diffusion language models know the answer before decoding. *arXiv preprint arXiv:2508.19982*, 2025.

[26] Tianyi Li, Mingda Chen, Bowei Guo, and Zhiqiang Shen. A survey on diffusion language models. *arXiv preprint arXiv:2508.10875*, 2025.

[27] 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. *arXiv preprint arXiv:2506.06295*, 2025.

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

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

[30] Yuxin Ma, Lun Du, Lanning Wei, Kun Chen, Qian Xu, Kangyu Wang, Guofeng Feng, Guoshan Lu, Lin Liu, Xiaojing Qi, et al. dinfer: An efficient inference framework for diffusion language models. *arXiv preprint arXiv:2510.08666*, 2025.

[31] Quan Nguyen-Tri, Mukul Ranjan, and Zhiqiang Shen. Attention is all you need for kv cache in diffusion llms. *arXiv preprint arXiv:2510.14973*, 2025.

[32] Shen Nie, Fengqi Zhu, Chao Du, Tianyu Pang, Qian Liu, Guangtao Zeng, Min Lin, and Chongxuan Li. Scaling up masked diffusion models on text. In *International Conference on Learning Representations*, volume 2025, pages 82974–82997, 2025.

[33] Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. *arXiv preprint arXiv:2502.09992*, 2025.

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

[35] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of machine learning and systems*, 5:606–624, 2023.

[36] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. OpenAI technical report, 2018.

[37] Liran Ringel, Ameen Ali, and Yaniv Romano. Dependency-guided parallel decoding in discrete diffusion language models. *arXiv preprint arXiv:2604.02560*, 2026.

[38] Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. *arXiv preprint arXiv:2406.07524*, 2024.

[39] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

[40] Jiaxin Shi, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis K Titsias. Simplified and generalized masked diffusion for discrete data. *arXiv preprint arXiv:2406.04329*, 2024.

[41] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

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

[43] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774*, 2024.

[44] Qingyan Wei, Yaojie Zhang, Zhiyuan Liu, Dongrui Liu, and Linfeng Zhang. Accelerating diffusion large language models with slowfast: The three golden principles. *arXiv preprint arXiv:2506.10848*, 2025.

[45] Chengyue Wu, Hao Zhang, Shuchen Xue, Shizhe Diao, Yonggan Fu, Zhijian Liu, Pavlo Molchanov, Ping Luo, Song Han, and Enze Xie. Fast-dllm v2: Efficient block-diffusion llm. In *International Conference on Learning Representations*, volume 2026, pages 128353–128370, 2026.

[46] Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. Fast-dllm: Training-free acceleration of diffusion llm by enabling kv cache and parallel decoding. *arXiv preprint arXiv:2505.22618*, 2025.

[47] Shutong Wu and Jiawei Zhang. Free draft-and-verification: Toward lossless parallel decoding for diffusion large language models. *arXiv preprint arXiv:2510.00294*, 2025.

[48] Tianyi Wu, Xiaoxi Sun, Yanhua Jiao, Yulin Li, Yixin Chen, Yun-Hao Cao, Yi-Qi Hu, and Zhuotao Tian. Dynamic-dllm: Dynamic cache-budget and adaptive parallel decoding for training-free acceleration of diffusion llm. In *The Fourteenth International Conference on Learning Representations*, 2026.

[49] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, volume 2024, pages 21875–21895, 2024.

[50] Zhihui Xie, Jiacheng Ye, Lin Zheng, Jiahui Gao, Jingwei Dong, Zirui Wu, Xueliang Zhao, Shansan Gong, Xin Jiang, Zhenguo Li, et al. Dream-coder 7b: An open diffusion language model for code. *arXiv preprint arXiv:2509.01142*, 2025.

[51] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[52] Ling Yang, Ye Tian, Bowen Li, Xinchen Zhang, Ke Shen, Yunhai Tong, and Mengdi Wang. Mmada: Multimodal large diffusion language models. *arXiv preprint arXiv:2505.15809*, 2025.

[53] Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7b: Diffusion large language models. *arXiv preprint arXiv:2508.15487*, 2025.

[54] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *Proceedings of Machine Learning and Systems*, 7, 2025.

[55] Ming Yin, Minshuo Chen, Kaixuan Huang, and Mengdi Wang. A theoretical perspective for speculative decoding algorithm. *Advances in Neural Information Processing Systems*, 37:128082–128117, 2024.

[56] Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. Flashattention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling. *arXiv preprint arXiv:2603.05451*, 2026.

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

[58] Kaiwen Zheng, Yongxin Chen, Hanzi Mao, Ming-Yu Liu, Jun Zhu, and Qinsheng Zhang. Masked diffusion models are secretly time-agnostic masked models and exploit inaccurate categorical sampling. *arXiv preprint arXiv:2409.02908*, 2024.

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

Appendix

Table of Contents

A Limitations
B Broader Impact
C Bounded-Deviation Guarantee for Flash-Verify
D Detailed Experiment Setup
E Statistical analysis
F Ablation of Generation and Prefill Length.
G More scalability analysis
H Sample Response

A Limitations

Flash-dLLM is evaluated on a single masked diffusion LLM (LLaDA-1.5) and a single GPU type (NVIDIA A100) across mathematical reasoning and code generation tasks. While the fused Triton kernel and Flash-Verify are architecture-agnostic in design, we have not yet validated them on other masked diffusion LLMs or on continuous-space diffusion language models, where the cache update patterns may differ. Similarly, our benchmarks focus on structured-output tasks; the behavior of confidence-aware decoding on open-ended generation (e.g., long-form writing or dialogue), where token-level confidence distributions tend to be flatter, is an interesting direction for future work. The hyperparameters γ and β_m are fixed throughout generation; an adaptive scheme that adjusts these based on running confidence statistics could further improve the throughput-accuracy trade-off. We view these as natural extensions and a potential future research direction.

B Broader Impact

This work targets inference-time efficiency for diffusion LLMs and does not introduce new training data, model architectures, or fine-tuning procedures. By reducing the computational and memory cost of dLLM decoding, Flash-dLLM lowers the hardware barrier to deploying these models, which could broaden access to non-autoregressive language generation for researchers and practitioners with limited GPU resources. Faster inference also reduces the energy consumption per generated sequence, contributing to more sustainable deployment of large language models. We do not modify safety filters or alignment mechanisms of the underlying models; the generation quality and any associated risks (e.g., producing harmful or biased content) are inherited from the base dLLM. We encourage users to apply standard content moderation and human oversight when deploying Flash-dLLM in user-facing applications, particularly in high-stakes domains.

C Bounded-Deviation Guarantee for Flash-Verify

Setup. We first fix the two distributions that Flash-Verify evaluates at a search position, following the draft and verify passes of Section 2.3, Figure 2 and Algorithm 2. Throughout, the quantity being controlled is the gap between committing several positions at once and committing them from the model's own chain-rule joint; this gap is the standard error term of parallel unmasking in masked diffusion models [46, 6, 37].

Draft pass. At denoising step t, the draft pass runs the model on the query set $\mathcal{Q}^t = \mathcal{T}^t \cup \mathcal{M}^t_{\beta_m}$ with every position in $\mathcal{M}^t_{\beta_m}$ held at [MASK]. Let $\mathbf{x}^t$ denote this input sequence. For each $i \in \mathcal{M}^t_{\beta_m}$ the draft pass yields the draft-pass distribution over the vocabulary V, which we call the *draft marginal*

$$q_i(x) := p_\theta\big(x^i = x \mid \mathbf{x}^t\big), \qquad x \in V, \tag{3}$$

from which the draft prediction and its confidence are $\hat{x}^i = \arg\max_x q_i(x)$ and $c^i = q_i(\hat{x}^i)$. The window is partitioned by confidence into the confident set $\mathcal{D}^t = \{i : c^i \geq \epsilon\}$, which is accepted directly, and the search set $\mathcal{S}^t = \mathcal{M}^t_{\beta_m} \setminus \mathcal{D}^t$, which is sent to verification. Note that q_i conditions on no other prediction made at step t: every position in $\mathcal{M}^t_{\beta_m}$, including $\mathcal{D}^t$ and the other search positions, is masked in $\mathbf{x}^t$. Committing $\mathcal{D}^t$ directly is covered by the analysis of confidence-aware

decoding [46, Thm. 1]: if n positions all have marginal confidence above $1-\varepsilon$ with $(n+1)\varepsilon \le 1$, the joint mode equals the product-of-marginals mode and the two distributions are within $\frac{3n-1}{2}\varepsilon$ in total variation. That result says nothing about $\mathcal{S}^t$, whose positions fail the confidence condition by construction, and it is for those positions that the verify pass and the guarantee below are needed.

Verify pass. Let $\mathbf{x}^t_+$ denote $\mathbf{x}^t$ with the confident positions committed to their drafts, $x^{\mathcal{D}^t} = \hat{x}^{\mathcal{D}^t}$; together with the previously decoded tokens this forms the tracked context $\mathcal{T}_v$. Write the search positions in causality order (draft confidence, descending) as $i_1, \dots, i_K$ with $K = |\mathcal{S}^t|$. The verify query duplicates $\mathcal{S}^t$ into a *draft view*, filled with $\hat{x}^{\mathcal{S}^t}$, and a *mask view*, filled with $[\mathrm{MASK}]$. Under the causal verify mask of Figure 2, the mask view at i_k attends to $\mathcal{T}_v$, to the draft view at $i_1, \dots, i_{k-1}$ and to the mask view at $i_k, \dots, i_K$; it does not attend to the draft view at i_k or at any later position. Every search position therefore enters its context exactly once, as a draft if it precedes i_k and as $[\mathrm{MASK}]$ otherwise, so the model output at the mask view of i_k is the mask-view distribution, which we call the *verify conditional*

$$p_{i_k}(x) := p_\theta\big(x^{i_k} = x \mid \mathbf{x}^t_+,\ \hat{x}^{i_1}, \dots, \hat{x}^{i_{k-1}}\big), \qquad x \in V, \tag{4}$$

i.e. the model's prediction for i_k when the positions before it in causality order are committed to their drafts and $i_k, \dots, i_K$ remain masked. This is exactly the input that sequential decoding in this order would present at its k-th step, and it is the same object that speculative decoding verifies against: the target model's conditional at a drafted position given the accepted tokens before it [24, 9, 55]. The mask-view prediction and confidence are $\tilde{x}^i = \arg\max_x p_i(x)$ and $\tilde{c}^i = p_i(\tilde{x}^i)$. Flash-Verify accepts position i according to

$$\mathrm{accept}(i) = \mathbb{I}\big[\hat{x}^i = \tilde{x}^i\big] \cdot \mathbb{I}\big[\tilde{c}^i \ge \gamma\big], \tag{5}$$

applied in causality order and stopped at the first rejection, so the committed block is the longest prefix $\mathcal{A} = \{i_1, \dots, i_m\}$ on which every position is accepted. An accepted token is simultaneously the mode of the draft marginal q_i and a γ-confident mode of the verify conditional p_i. On acceptance the model commits the agreed token, and we write the delivered distribution at position i as the point mass $\pi_i(x) := \mathbb{I}[x = \tilde{x}^i]$. For two distributions u, v over a finite set, total variation distance is $\mathrm{TV}(u, v) := \frac{1}{2}\sum_x |u(x) - v(x)|$.

Assumption 1. *The reference for the committed block is the model's own sequential distribution in causality order: at position i_k it is the verify conditional p_{i_k} of Eq.* (4)*, and over the block it is the chain-rule joint $p_{\mathcal{A}}(x^{i_1}, \dots, x^{i_m}) := \prod_{k=1}^m p_\theta\big(x^{i_k} \mid \mathbf{x}^t_+, x^{i_1}, \dots, x^{i_{k-1}}\big)$, which is the distribution one-token-per-step decoding of the same positions in the same order samples from. Deviation is measured against this model distribution rather than the data distribution.*

Theorem 1 (Per-token deviation)**.** *Under Assumption 1, for every $i \in \mathcal{A}$,*

$$\mathrm{TV}(\pi_i, p_i) = 1 - \tilde{c}^i \le 1 - \gamma. \tag{6}$$

Proof. Let $m := \tilde{x}^i = \arg\max_x p_i(x)$, so that $p_i(m) = \tilde{c}^i$ and $\pi_i(x) = \mathbb{I}[x = m]$. By the definition of total variation,

$$\mathrm{TV}(\pi_i, p_i) = \tfrac{1}{2}\sum_{x \in V} \big|\pi_i(x) - p_i(x)\big| = \tfrac{1}{2}\Big(\big|\pi_i(m) - p_i(m)\big| + \sum_{x \ne m} \big|\pi_i(x) - p_i(x)\big|\Big). \tag{7}$$

At $x = m$ we have $\pi_i(m) = 1$, hence $|\pi_i(m) - p_i(m)| = 1 - p_i(m)$. At every $x \ne m$ we have $\pi_i(x) = 0$, hence $|\pi_i(x) - p_i(x)| = p_i(x)$, and since p_i is a probability distribution, $\sum_{x \ne m} p_i(x) = 1 - p_i(m)$. Combining,

$$\mathrm{TV}(\pi_i, p_i) = \tfrac{1}{2}\big[(1 - p_i(m)) + (1 - p_i(m))\big] = 1 - p_i(m) = 1 - \tilde{c}^i. \tag{8}$$

The acceptance rule fires only when $\tilde{c}^i \ge \gamma$, therefore $\mathrm{TV}(\pi_i, p_i) = 1 - \tilde{c}^i \le 1 - \gamma$. □

Corollary 1 (Block deviation)**.** *Let $\mathcal{A} = \{i_1, \dots, i_m\}$ be the block committed by a single verify pass, $\pi_{\mathcal{A}} := \prod_{k=1}^m \pi_{i_k}$ the delivered distribution over $\mathcal{A}$, and $p_{\mathcal{A}}$ the chain-rule joint of Assumption 1. Then*

$$\mathrm{TV}(\pi_{\mathcal{A}}, p_{\mathcal{A}}) = 1 - \prod_{k=1}^m \tilde{c}^{\,i_k} \le 1 - \gamma^m \le m\,(1 - \gamma). \tag{9}$$

Proof. Every accepted token equals its draft, $\tilde{x}^{i_k} = \hat{x}^{i_k}$, so $\pi_{\mathcal{A}}$ is the point mass at $\hat{x}^{\mathcal{A}} = (\hat{x}^{i_1}, \ldots, \hat{x}^{i_m})$. The computation in the proof of Theorem 1 applies verbatim to any point mass δ_z and any distribution p on a finite set and gives $\mathrm{TV}(\delta_z, p) = 1 - p(z)$; hence $\mathrm{TV}(\pi_{\mathcal{A}}, p_{\mathcal{A}}) = 1 - p_{\mathcal{A}}(\hat{x}^{\mathcal{A}})$. Evaluating the chain rule at $\hat{x}^{\mathcal{A}}$, the k-th factor is $p_\theta(\hat{x}^{i_k} \mid \mathbf{x}_+^t, \hat{x}^{i_1}, \ldots, \hat{x}^{i_{k-1}}) = p_{i_k}(\hat{x}^{i_k}) = p_{i_k}(\tilde{x}^{i_k}) = \tilde{c}^{i_k}$ by Eq. (4); the conditioning tokens $\hat{x}^{i_1}, \ldots, \hat{x}^{i_{k-1}}$ all lie in the accepted prefix, so no rejected draft enters any factor. Thus $p_{\mathcal{A}}(\hat{x}^{\mathcal{A}}) = \prod_k \tilde{c}^{i_k} \geq \gamma^m$, and $1 - \gamma^m \leq m(1-\gamma)$ by Bernoulli's inequality. □

Remark 1 (What the reference is)**.** *Because the verify mask is causal in causality order, $p_{\mathcal{A}}$ is the model's genuine joint over the block, not a product of marginals: the verified tokens incur no parallel-unmasking dependence error, and the bound is an equality in the mask-view confidences. The dependence term bounded in KL by [6] and in total variation by [37] therefore concerns only $\mathcal{D}^t$, which is committed from marginals and handled by [46, Thm. 1]. Stopping at the first rejection is what keeps $\mathcal{A}$ a prefix, so that every conditional in $p_{\mathcal{A}}$ conditions only on accepted tokens.*

Remark 2 (Why γ gates the mask view)**.** *The bound is controlled by γ because acceptance thresholds the confidence of p_i, the distribution the token is actually committed from. The confidence threshold ϵ instead gates $c^i = q_i(\hat{x}^i)$, computed with all of $\mathcal{M}^t_{\beta_m}$ masked [46]; q_i ignores the other positions decoded in the same step, so $\hat{x}^i$ can be its mode yet carry little mass under p_i, and for a search token $c^i < \epsilon$ by construction. Hence lowering ϵ is not equivalent to verifying: only the latter carries the $1 - \gamma$ guarantee against p_i. In the language of speculative decoding, the rule is a deterministic, biased acceptance [55]: the agreement test alone is lossless with respect to greedy decoding from p_i, as in the auto-speculative verifier of [2], while γ trades a bounded distribution bias of $1 - \tilde{c}^i$ for more accepted tokens, the trade-off swept in Figure 5a.*

D Detailed Experiment Setup

Setup. The model, benchmarks, baselines, and default hyper-parameters are described in Section 3.1. We test at generation lengths 256 and 512 to study scalability.

Evaluation Framework and Metrics. Our evaluation protocol comprehensively assesses both inference efficiency and model performance across various tasks. To ensure standardization and reproducibility, we conduct all task-specific evaluations using the `lm-eval-harness` library [14]. Inference speed is measured by throughput in tokens per second (t/s), calculated as the average number of tokens generated by the model over the entire sequence until it produces an end-of-sequence (`<eos>`) token. We maintain consistency with Fast-dLLM [46] in our calculation methodology to ensure comparable speed benchmarks. Task-specific performance is evaluated using established metrics suitable for each benchmark. For GSM8K [11], we report 5-shot `flexible_extract` exact match accuracy. For the MATH dataset [17], we report the 4-shot `math_verify` score using the `minerva_math` variant. For HumanEval [10], we evaluate 0-shot accuracy using a post-processing script consistent with the Fast-dLLM implementation to ensure fair comparison. Finally, for MBPP [5], we report the 3-shot `pass@1` metric.

Hyper-parameters: The hyper-parameters used for Flash-dLLM are presented in Table 3. Specifically,

- We set the basic block size to $\beta = 16$, corresponding to the smallest block used in our Triton implementation.
- For confidence-aware decoding, we use $\epsilon = 0.9$, following the optimal setting reported by Fast-dLLM.
- We use a default masked-window size of $\beta_m = 64$ across all settings to balance accuracy and decoding speed.
- We set $\gamma = 0.8$ for GSM8K and MBPP and $\gamma = 0.85$ for MATH and HumanEval.
- We use a batch size of 32 in all experiments to maximize GPU utilization.

E Statistical analysis

Table 4 examines the joint effect of the verification threshold γ and the track budget β_t on accuracy and throughput. The results show a consistent accuracy–efficiency trade-off: increasing either

Table 3: The hyper-parameters of Flash-dLLM under various settings.

Model	Benchmark	Gen Length	Tracking budget β_t	Window size β_m	Flash-Verify γ	Batch size
LLaDA-1.5	GSM8K (5-shot)	256	64	64	0.8	32
		512	64	64	0.8	32
	MATH (4-shot)	256	64	64	0.85	32
		512	64	64	0.85	32
	Humaneval (0-shot)	256	64	64	0.85	32
		512	64	64	0.85	32
	MBPP (3-shot)	256	48	64	0.8	32
		512	48	64	0.8	32

Table 4: Mean accuracy (%) and throughput over five random seeds. Accuracy is shown on the first line and throughput is shown in blue on the second line. Values are mean ± standard deviation.

	Track budget β_t			
γ	48	64	80	96
0.60	80.23 ± 0.46 277.80 ± 1.18	80.76 ± 0.21 261.54 ± 1.00	81.52 ± 0.50 248.80 ± 0.70	81.42 ± 0.41 235.82 ± 0.76
0.70	80.74 ± 0.66 252.20 ± 2.52	81.27 ± 0.65 234.86 ± 4.94	82.00 ± 0.36 226.66 ± 3.28	82.35 ± 0.13 214.78 ± 2.18
0.75	80.94 ± 0.81 235.60 ± 1.36	82.14 ± 0.47 221.68 ± 0.71	82.52 ± 0.45 209.76 ± 1.29	82.56 ± 0.91 208.08 ± 0.51
0.80	81.64 ± 0.54 221.06 ± 3.58	81.73 ± 0.74 208.40 ± 2.64	82.15 ± 0.47 197.08 ± 1.54	82.47 ± 0.47 196.98 ± 0.84
0.85	81.46 ± 1.03 211.32 ± 0.69	82.71 ± 0.51 197.60 ± 0.70	82.76 ± 0.84 187.28 ± 0.75	83.15 ± 0.17 185.70 ± 0.61
0.90	81.55 ± 0.49 206.22 ± 0.50	82.09 ± 0.53 192.90 ± 0.75	82.64 ± 0.41 182.16 ± 0.68	82.75 ± 0.62 171.86 ± 0.33
1.00	81.79 ± 0.64 165.14 ± 0.83	82.24 ± 0.43 149.60 ± 0.47	82.96 ± 0.51 138.28 ± 0.37	83.41 ± 0.50 131.58 ± 1.41

parameter generally improves accuracy while reducing decoding throughput. At a fixed γ, increasing β_t from 48 to 96 improves accuracy by approximately 0.83–1.69 percentage points, but decreases throughput by about 10.9%–20.3%. Similarly, increasing γ from 0.60 to 1.00 yields an accuracy gain of 1.44–1.99 percentage points across the evaluated track budgets, accompanied by a throughput reduction of approximately 40.6%–44.4%. Although a few neighboring configurations exhibit small non-monotonic variations, these differences are generally comparable to the reported standard deviations and do not alter the overall trend.

The highest-throughput configuration is obtained with $\gamma = 0.60$ and $\beta_t = 48$, reaching 277.80 tokens/s at 80.23% accuracy. In contrast, the highest accuracy of 83.41% is achieved with $\gamma = 1.00$ and $\beta_t = 96$, where throughput decreases to 131.58 tokens/s. Thus, moving from the fastest to the most accurate configuration improves accuracy by 3.18 percentage points while reducing throughput by approximately 52.6%. Intermediate settings provide more balanced operating points. In particular, $\gamma = 0.85$ and $\beta_t = 96$ achieve 83.15% accuracy at 185.70 tokens/s, remaining only 0.26 percentage points below the maximum accuracy while providing approximately 41.1% higher throughput. The relatively small standard deviations across five random seeds—at most 1.03 percentage points for accuracy and 4.94 tokens/s for throughput—also indicate that the observed trade-off is consistent across runs. Overall, γ and β_t offer complementary controls for selecting either a throughput-oriented or accuracy-oriented operating point.

F Ablation of Generation and Prefill Length.

Table 5 examines the sensitivity of the two decoding strategies to generation length and prefill length. As shown in Table 5a, Flash-Cache with Flash-Verify consistently provides higher throughput than confidence-aware decoding across all generation lengths. Its throughput advantage ranges from 1.38× to 1.41×, corresponding to absolute improvements of 47.3–58.0 tokens/s. Both methods reach their highest throughput at a generation length of 256, attaining 140.4 tokens/s for confidence-aware decoding and 198.4 tokens/s for Flash-Verify. Beyond 256 tokens, throughput gradually decreases, while accuracy remains nearly unchanged. For example, increasing the generation length from 256

Table 5: Ablation of generation and prefill lengths.

(a) Impact of generation length on LLaDA-1.5 for GSM8K (5-shot) with a batch size of 16.

Approach	Metric	Len. 128	Len. 256	Len. 512	Len. 1024
Flash-Cache + Confidence	Accuracy	79.03	82.36	82.24	82.29
	Throughput	126.7 (1.00×)	140.4 (1.00×)	131.8 (1.00×)	125.7 (1.00×)
	Tokens/step	2.5	2.8	2.8	2.9
Flash-Cache + Flash-Verify	Accuracy	79.55	81.90	81.73	81.72
	Throughput	174.5 (1.38×)	198.4 (1.41×)	186.2 (1.41×)	173.0 (1.38×)
	Tokens/step	5.6	5.6	5.6	5.6

(b) Impact of prefill length on LLaDA-1.5 for GSM8K-512 with a batch size of 16.

Approach	Metric	1-shot	3-shot	5-shot	8-shot
Flash-Cache + Confidence	Accuracy	80.00	82.27	82.24	82.25
	Throughput	155.8 (1.00×)	145.4 (1.00×)	131.8 (1.00×)	123.1 (1.00×)
	Tokens/step	2.8	2.8	2.8	2.8
Flash-Cache + Flash-Verify	Accuracy	80.12	81.73	81.73	81.73
	Throughput	234.4 (1.50×)	209.6 (1.44×)	186.2 (1.41×)	165.7 (1.35×)
	Tokens/step	5.8	5.7	5.7	5.6

Table 6: Effect of batch size on throughput. Using LLaDA-1.5 for GSM8K-512 (5-shot). Speedup is measured relative to Flash-Cache with greedy decoding at the same batch size.

Approach	ACC	Tok./step	Throughput						
			B=1	B=2	B=4	B=8	B=16	B=24	B=32
Flash-Cache + Greedy	82.94	1.0	18.5 (1.0×)	28.9 (1.0×)	37.7 (1.0×)	46.3 (1.0×)	51.3 (1.0×)	53.6 (1.0×)	55.0 (1.0×)
Flash-Cache + Confidence	82.87	2.8	51.7 (2.8×)	78.0 (2.7×)	102.4 (2.7×)	120.5 (2.6×)	131.8 (2.6×)	136.4 (2.5×)	139.5 (2.5×)
Flash-Cache + Flash-Verify	**83.02**	5.7	56.0 (3.0×)	90.9 (3.1×)	131.3 (3.5×)	164.8 (3.6×)	186.2 (3.6×)	195.5 (3.6×)	199.8 (3.6×)

to 1024 reduces throughput by approximately 10.5% for confidence-aware decoding and 12.8% for Flash-Verify, without providing a corresponding accuracy improvement.

Table 5b shows that increasing the number of in-context examples, and consequently the prefill length, reduces throughput for both methods. From 1-shot to 8-shot prompting, confidence-aware throughput decreases from 155.8 to 123.1 tokens/s, a reduction of approximately 21.0%. Flash-Verify decreases from 234.4 to 165.7 tokens/s, corresponding to a reduction of approximately 29.3%. Nevertheless, Flash-Verify remains faster in every setting, providing speedups of $1.50\times$, $1.44\times$, $1.41\times$, and $1.35\times$ for the 1-, 3-, 5-, and 8-shot settings, respectively. The decreasing relative speedup suggests that the computational cost of processing a longer prefix affects Flash-Verify more strongly, although it retains an absolute advantage of 42.6–78.6 tokens/s.

G More scalability analysis

Analysis. Table 6 evaluates the throughput scaling of the three decoding strategies as the batch size increases. Throughput improves monotonically for all methods, although the gains gradually diminish at larger batch sizes. From $B = 1$ to $B = 32$, greedy decoding increases from 18.5 to 55.0 tokens/s, corresponding to a $2.97\times$ increase due to batch scaling. Confidence-aware decoding rises from 51.7 to 139.5 tokens/s, or $2.70\times$, while Flash-Verify increases from 56.0 to 199.8 tokens/s, corresponding to the largest batch-scaling improvement of $3.57\times$.

Flash-Verify achieves the highest throughput at every batch size. Relative to greedy decoding under the same batch configuration, its speedup increases from $3.0\times$ at $B = 1$ to approximately $3.6\times$ for $B \geq 8$. Confidence-aware decoding provides a smaller speedup of approximately $2.5\times$–$2.8\times$. The advantage of Flash-Verify over confidence-aware decoding also grows with batch size: it is only 8.3% faster at $B = 1$, but becomes approximately 43.2% faster at $B = 32$. This trend indicates that Flash-Verify makes more effective use of the additional parallel computation available at larger batch sizes.

The throughput improvements are consistent with the number of tokens decoded per step. Flash-Verify accepts 5.7 tokens per step, compared with 2.8 for confidence-aware decoding and 1.0 for greedy decoding. Throughput begins to saturate beyond $B = 16$: at this batch size, Flash-Verify already reaches 186.2 tokens/s, or approximately 93.2% of its throughput at $B = 32$. Therefore, $B = 16$ provides a favorable efficiency point, capturing most of the maximum throughput while requiring only half the batch size.

H Sample Response

In the following, we present several examples of actual generation results produced by our approach under different parameter settings.

Table 7: Example LLaDA-1.5 (without cache) — A qualitative comparison of different decoding methods.

Prompt: *Amy is taking a history test. She correctly answers 80% of the multiple-choice questions, 90% of the true/false questions, and 60% of the long-answer questions. The multiple-choice and true/false questions are worth 1 point each, and the long answer questions are worth 5 points each. How many points does Amy score if there are 10 multiple-choice questions, 20 true/false questions, and 5 long answer questions?*

Greedy	Confidence-aware	Flash-Verify
First find the number of multiple-choice questions Amy answers correctly: 80% * 10 questions = «10*.80=8»8 questions Then find the number of true/false questions Amy answers correctly: 90% * 20 questions = «20*.90=18»18 questions Then find the number of long-answer questions Amy answers correctly: 60% * 5 questions = «50*.60=3»3 questions Then find the total points Amy gets from multiple-choice questions: 8 questions * 1 point/question = «8*1=8»8 points Then find the total points Amy gets from true/false questions: 18 questions * 1 point/question = «18*1=18»18 points Then find the total points Amy gets from long-answer questions: 3 questions * 5 points/question = «3*5=15»15 points Then add up the points from each type of question to find her total score: 8 points + 18 points + 15 points = «8+18+15=41»41 points $\boxed{41}$ **Steps: 283, Tokens: 283, Time:64.7s**	First find the number of multiple-choice questions Amy answers correctly: 80% * 10 questions = «80*.10=8»8 questions Then find the number of true/false questions Amy answers correctly: 90% * 20 questions = «90*.20=18»18 questions Then find the number of long-answer questions Amy answers correctly: 60% * 5 questions = «60*.05=3»3 questions Then find the total points Amy gets from multiple-choice questions: 8 questions * 1 point/question = «8*1=8»8 points Then find the total points Amy gets from true/false questions: 18 questions * 1 point/question = «18*1=18»18 points Then find the total points Amy gets from long-answer questions: 3 questions * 5 points/question = «3*5=15»15 points Then add up the points from each type of question to find her total score: 8 points + 18 points + 15 points = «8+18+15=41»41 points $\boxed{41}$ **Steps: 66, Tokens: 277, Time: 17.2s**	First find the number of multiple-choice questions Amy answers correctly: 10 questions * 80% = «10*0.8=8»8 questions Then find the number of true/false questions Amy answers correctly: 20 questions * 90% = «20*0.9=18»18 questions Then find the number of long-answer questions Amy answers correctly: 5 questions * 60% = «5*0.6=3»3 questions Then find the total points from the multiple-choice questions: 8 questions * 1 point/question = «8*1=8»8 points Then find the total points from the true/false questions: 18 questions * 1 point/question = «18*1=18»18 points Then find the total points from the long-answer questions: 3 questions * 5 points/question = «3*5=15»15 points Then add up the points from each type of question to find the total score: 8 points + 18 points + 15 points = «8+18+15=41»41 points $\boxed{41}$ **Steps: 30, Tokens: 276, Time: 13.1s**

Table 8: Example Flash-dLLM: A qualitative comparison of different decoding methods, LLaDA-1.5

Prompt: *Rong has been saving 20 coins in his piggy bank every month. Neil has been saving 2/5 times more coins in his piggy bank per month than Rong. How many coins are they having ten years after they started their savings?*

Greedy	Confidence-aware	Flash-Verify
Rong saves 20 coins per month, so in one year, he saves 20 * 12 = «20*12=240»240 coins. Neil saves 2/5 times more coins than Rong, so he saves 20 + (2/5) * 20 = 20 + 8 = 28 coins per month. In one year, Neil saves 28 * 12 = «28*12=336»336 coins. In ten years, Neil saves 336 * 10 = «336*10=3360»3360 coins. Together, Rong and Neil have 2400 + 3360 = «2400+3360=5760»5760 coins. $\boxed{5760}$ **Steps: 224, Tokens: 224, Time:15.2s**	Rong saves 20 coins per month, so in one year, he saves 20 * 12 = «20*12=240»240 coins. Neil saves 2/5 times more coins than Rong, so he saves 20 + (2/5) * 20 = 20 + 8 = 28 coins per month. In one year, Neil saves 28 * 12 = «28*12=336»336 coins. In ten years, Neil saves 336 * 10 = «336*10=3360»3360 coins. Together, Rong and Neil have saved 240 + 3360 = «240+3360=3600»3600 coins. $\boxed{3600}$ **Steps: 73, Tokens: 222, Time: 7.8s**	Rong saves 20 coins per month, so in one year, he saves 20 * 12 = «20*12=240»240 coins. Neil saves 2/5 times more coins than Rong, so he saves 20 + (2/5 * 20) = 20 + 8 = 28 coins per month. In one year, Neil saves 28 * 12 = «28*12=336»336 coins. In ten years, Neil saves 336 * 10 = «336*10=3360»3360 coins. Together, Rong and Neil have saved 240 + 3360 = «240+3360=3600»3600 coins. $\boxed{3600}$ **Steps: 34, Tokens: 219, Time: 5.7s**

Table 9: Example Flash-Verify (without cache) — A qualitative comparison of different γ values, LLaDA-1.5

Prompt: *A nurses' station orders bandages in bulk packs of 50. On the first day, the nurses used 38 bandages and ordered one bulk pack of bandages. On the second day, they used ten fewer bandages. On the third day, they ordered two bulk packs of bandages and only used half a pack. They had 78 bandages left at the end of the third day. How many bandages did they start with on the first day?*

$\gamma = 0.8$	$\gamma = 0.9$	$\gamma = 0.95$
First, find the total number of bandages used on the first day: 38 bandages + 50 bandages = «38+50=88»88 bandages Then find the total number of bandages used on the second day: 88 bandages - 10 bandages = «88-10=78»78 bandages Then find the total number of bandages used on the third day: 2 * 50 bandages = «2*50=100»100 bandages Then find the total number of bandages used on the third day: 100 bandages / 2 = «100/2=50»50 bandages Then find the total number of bandages used over the three days: 88 bandages + 78 bandages + 50 bandages = «88+78+50=216»216 bandages Then add the number of bandages left at the end of the third day to find the total number of bandages started: 216 bandages + 78 bandages = «216+78=294»294 bandages Then 294 bandages $\boxed{294}$ **Steps: 38, Tokens: 298, Time:11.3s**	First, let's total how many bandages were used on the first day and how many were ordered: - Used: 38 bandages - Ordered: 1 bulk pack = 50 bandages So, the total number of bandages at the end of the first day is: $$38 + 50 = 88$$ Next, let's determine how many bandages were used on the second day: - Used: 38 - 10 = 28 bandages So, the total number of bandages at the end of the second day is: $$88 - 28 = 60$$ Now, let's determine how many bandages were used on the third day: - Ordered: 2 bulk packs = 100 bandages - Used: 1/2 = 50 bandages So, the total number of bandages at the end of the third day is: $$60 + 100 - 50 = 110$$ We know that they had 78 bandages left at the end of the third day, so the total number of bandages they started with on the first day is: $$110 + 78 = 188$$ Therefore, they started with 188 bandages on the first day. $\boxed{188}$ **Steps: 71, Tokens: 330, Time: 22.0s**	First, find the total number of bandages used on the first day: 38 bandages + 50 bandages = «38+50=88»88 bandages Then find the total number of bandages used on the second day: 88 bandages - 10 bandages = «88-10=78»78 bandages Then find the total number of bandages used on the third day: 2 * 50 bandages = «2*50=100»100 bandages Then find the total number of bandages used over the three days: 88 bandages + 78 bandages + 100 bandages = «88+78+100=266»266 bandages Then add the number of bandages left at the end of the third day to the total number of bandages used to find the total number of bandages at the start of the first day: 266 bandages + 78 bandages = «266+78=344»344 bandages $\boxed{344}$ **Steps: 57, Tokens: 271, Time: 19.8s**

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The main claims made in the abstract and introduction accurately reflect the paper's contributions and scope.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Discussed in Appendix A.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

Justification: Appendix C states the assumption (Assumption 1) and gives complete proofs of the bounded-deviation guarantee for Flash-Verify (Theorem 1, Corollary 1); the other contributions are empirical.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: All details are provided in Section 3.1 and Appendix D.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: The setup is described in Section 3 and Appendix D. The code is released at `https://github.com/VILA-Lab/Flash-dLLM`.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Discussed in Section 3.1 and Appendix D.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: The main comparison tables report single runs, following the evaluation convention of prior diffusion-LLM acceleration work. Mean and standard deviation over five random seeds are reported for the γ–β_t sweep and Pareto analysis (Appendix E, Table 4, Fig. 5).

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: All experiments run on a single NVIDIA A100 80GB GPU (Section 3.1).

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Discussed in Appendix B.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.

- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A].

    Justification: Not Applicable for the paper.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Yes, all the papers, models and datasets are properly cited.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The code is released at `https://github.com/VILA-Lab/Flash-dLLM` under the Apache-2.0 license.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: Not Applicable for this paper.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: Not Applicable for this paper.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: LLM was used only to help solely polish the writing of the paper, while all method, idea and experiments were prepared and carried out entirely by the authors.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.