SpecFold: Folding Cross-Branch Redundancy for Faster Speculative Decoding in Diffusion Language Models

Anonymous ACL submission

Abstract

Diffusion large language models (DLLMs) generate text through iterative block denoising, and speculative decoding accelerates this process by verifying a main branch together with multiple draft branches in a single forward pass. While prior DLLM acceleration methods primarily exploit temporal redundancy across denoising steps, we observe that speculative verification introduces a second redundancy axis within the verification pass itself: draft branches inherit most tokens from their parents and modify only a small set of newly unmasked positions, causing large portions of their hidden states to remain highly similar across layers and token positions. We propose SPECFOLD, an algorithm that exploits this cross-branch redundancy in speculative DLLM decoding. SPECFOLD performs token-level pre-QKV similarity gating and folds redundant draft branch computation into parent drafts, while recomputing divergent positions through folded attention and feed-forward execution that preserves the speculative verification structure. SPECFOLD is orthogonal to temporal caching and compatible with existing main-plus-draft speculative policies. Across four representative DLLMs (Fast-dLLM-v2-7B, Fast-dLLM-v2-1.5B, and the Fast-dLLM-v1 variants of LLaDA-8B and Dream-7B) on five standard benchmarks, SPECFOLD achieves up to 62% FLOPs reduction over Spiffy while largely preserving task performance.

1 Introduction

Diffusion large language models (DLLMs) generate text by iteratively denoising token blocks, exposing block-level parallelism that autoregressive (AR) decoding cannot exploit (Nie et al., 2025; Ye et al., 2025; Wu et al., 2025a; Arriola et al., 2025). The paradigm has scaled from small-scale block-diffusion prototypes (Arriola et al., 2025), through dense models such as LLaDA, Dream, and Fast-dLLM (Nie et al., 2025; Ye et al., 2025; Wu et al., 2025a), to larger sparse and token-editing variants (Zhu et al., 2025; Bie et al., 2025, 2026). However, high-quality DLLM generation still requires repeated refinement of the active block. After optimizations such as block-wise decoding, confidence-based token revealing, adaptive caching, dynamic cache eviction, and layer skipping (Liu et al., 2025; Ma et al., 2025; Song et al., 2026; Goel et al., 2026; Wu et al., 2025a; Sun et al., 2026), the remaining active-block computation is often dominated by repeated forward passes that underutilize modern GPU compute in optimized settings (Gao et al., 2025; Wu et al., 2025a).

Speculative decoding offers a complementary route to reducing the number of denoising calls. In autoregressive models, speculative decoding accelerates generation by drafting candidate future tokens and verifying them with the target model (Leviathan et al., 2023; Chen et al., 2023; Cai et al., 2024; Li et al., 2024). Recent work adapts this principle to DLLMs by constructing draft branch states and verifying them together with the main branch in a single batched forward pass (Agrawal et al., 2025; Li et al., 2025; Gao et al., 2025). This adaptation is structurally different from autoregressive speculation: DLLM attention is bidirectional, so each draft is a full block state rather than a next-token continuation, and draft states across denoising steps form a directed graph of related masked-token configurations rather than a causal tree. Existing speculative DLLM methods therefore verify one main branch and B draft branches in one pass. If a draft matches the subsequent main branch trajectory, its precomputed logits can be reused, allowing the decoder to skip one or more future denoising calls.

While speculative decoding reduces the number of function evaluations, it also introduces a new bottleneck: the verification pass densely evaluates all draft branches. The central observation of this paper is that this dense verification pass

contains substantial *cross-branch redundancy*. By construction, a draft inherits almost all tokens from its parent and differs only at a small set of newly unmasked positions. Consequently, most token representations in the child draft remain nearly identical to the corresponding parent representations across many layers. Our profiling on Fast-dLLM-v1 LLaDA-8B (Nie et al., 2025; Wu et al., 2025b) confirms that more than 80% of measured layer–token–step triples have large parent–child pre-QKV hidden-state cosine similarity, with divergence concentrated near recently unmasked positions. Moreover, parent–child similarity is consistently higher than draft–main similarity because the parent is the draft from which the draft branch directly inherits most of its unmasked tokens. Existing speculative DLLM systems nevertheless process every draft branch densely, recomputing layers for representations that are already available from the parent draft (Agrawal et al., 2025; Li et al., 2025; Gao et al., 2025).

To exploit this redundancy, we propose SPECFOLD, an algorithm for folded speculative DLLM decoding. SPECFOLD operates before the QKV projection of each Transformer layer. For every parent–child draft pair, it measures token-level hidden-state similarity and folds a child token into its parent when the similarity exceeds a threshold. Folded positions reuse parent draft computation, while divergent positions are recomputed. Because self-attention couples all token positions, SPECFOLD does not simply drop tokens or copy outputs. Instead, it evaluates the layer on the folded hidden states through folded attention and feed-forward execution, preserving the verification structure required by speculative decoding.

SPECFOLD is orthogonal to prior DLLM acceleration methods. Temporal caching and layer skipping reuse computation across denoising steps along a single trajectory (Liu et al., 2025; Ma et al., 2025; Song et al., 2026; Goel et al., 2026; Sun et al., 2026), whereas speculative DLLM methods reduce the number of denoising calls by verifying drafts in parallel (Agrawal et al., 2025; Li et al., 2025; Gao et al., 2025). SPECFOLD instead removes redundant computation *within* each speculative verification pass, across the drafts that are already evaluated together. It therefore composes with existing main-plus-draft policies such as Spiffy (Agrawal et al., 2025) without retraining or modifying draft construction. Our contributions are threefold:

- We identify *cross-branch activation redundancy* as a new efficiency opportunity in speculative DLLM decoding. Unlike temporal redundancy across denoising steps, this redundancy arises within a single verification pass.
- We introduce SPECFOLD, a folded speculative decoding algorithm that performs token-level pre-QKV similarity gating, layer-wise hidden-state reuse, folded self-attention correction, folded feed-forward execution, and speculative-cache-compatible output generation. The strategy is agnostic to the underlying draft-construction policy and applies to current DLLM speculative decoding methods.
- We show that SPECFOLD is compatible with existing speculative DLLM policies such as Spiffy (Agrawal et al., 2025) and improves efficiency across four DLLMs and five standard benchmarks, reducing speculative verification computation substantially while preserving comparable task performance.

2 Background

2.1 Masked Diffusion LLMs and Temporal Acceleration

Masked diffusion large language models (DLLMs) generate text by reversing a masking process over token blocks (Nie et al., 2025; Ye et al., 2025; Arriola et al., 2025). During training, the model observes a partially masked sequence and learns to reconstruct the original tokens. During inference, generation proceeds block by block: for the current active block, the decoder starts from masked positions and repeatedly predicts token distributions, reveals a subset of positions, and refines the remaining masked positions until the block is complete. Unlike autoregressive decoding, a single DLLM denoising step can update multiple token positions in parallel, exposing block-level parallelism.

We write k for the active block index and t for the denoising timestep within that block. Let N denote the active-block length, and let $X_k^t \in \mathcal{V}^N$ denote the current block state, whose entries are either vocabulary tokens or the mask token `[M]`. The completed prefix, prompt, and previously finalized blocks are treated as context C_k. A DLLM forward pass can be viewed as a map

$$F_\theta(C_k, X_k^t) \to p_\theta(\cdot \mid C_k, X_k^t), \qquad (1)$$

which returns token distributions for the masked positions in the active block. A reveal rule then

selects one or more high-confidence positions to unmask, producing the next state X_k^{t-1}.

This iterative refinement is the main source of DLLM inference cost. Recent DLLM systems reduce this cost by improving the denoising schedule or by reusing computation across adjacent denoising states. These include block-wise decoding, confidence-based token revealing, adaptive activation caching, dynamic cache eviction, and layer skipping (Liu et al., 2025; Ma et al., 2025; Song et al., 2026; Goel et al., 2026; Wu et al., 2025a; Sun et al., 2026). Such methods exploit *temporal redundancy*: adjacent states X_k^t and X_k^{t-1} usually differ at only a few newly revealed positions, so parts of the computation along a single denoising trajectory can be reused. However, even after temporal acceleration, the active block still requires repeated model evaluations, and optimized DLLM decoding can remain limited by repeated forward-pass overhead and poor hardware utilization (Gao et al., 2025; Wu et al., 2025a). Speculative DLLM decoding addresses this remaining cost by attempting to skip future denoising calls.

2.2 Speculative Decoding in DLLMs

Speculative decoding accelerates generation by proposing candidate future states and verifying them with the target model. In autoregressive language models, drafts are causal next-token continuations that are accepted or rejected by the target model (Leviathan et al., 2023; Chen et al., 2023; Cai et al., 2024; Li et al., 2024). Adapting this idea to DLLMs requires a different structure because DLLM attention is bidirectional: a draft is not a next-token continuation, but a full active-block state with a particular pattern of unmasked and masked positions.

At a speculative DLLM verification step, the decoder constructs a batch of $B+1$ branch states: one main branch $X_{0,k}^t$ and B draft branches $\{X_{i,k}^t\}_{i=1}^B$. The branches form a directed draft graph. Each draft branch i has a parent draft $\mathrm{par}(i) \in \{0, \ldots, i-1\}$, inherits the parent draft state, and differs from it by at most S_t newly unmasked positions. The target DLLM then verifies all branches in a single batched forward pass:

$$p_{i,k}^t = F_\theta(C_k, X_{i,k}^t), \qquad i = 0, 1, \ldots, B. \quad (2)$$

The resulting pairs $\{(X_{i,k}^t, p_{i,k}^t)\}_{i=1}^B$ are stored in a speculative cache. If a later main-branch state matches one of the cached draft states, the decoder can reuse the corresponding precomputed logits and skip an additional DLLM call. In lossless speculative DLLM schemes, this cache reuse preserves the target-model trajectory because the reused logits are those produced by the same target model on the same block state (Agrawal et al., 2025; Li et al., 2025; Gao et al., 2025).

Existing speculative DLLM methods differ in how they construct draft states and acceptance policies, but they share the same computational pattern: the verification pass evaluates all drafts densely (Agrawal et al., 2025; Li et al., 2025; Gao et al., 2025). This dense batched verification reduces the number of denoising calls, but it introduces a new source of redundant computation. A child draft usually inherits almost all tokens from its parent and changes only a small set of newly unmasked positions. Nevertheless, standard speculative verification recomputes attention and feed-forward layers for every token in every draft.

This paper targets the redundancy left unexploited by temporal acceleration and dense speculative verification. Temporal methods reuse computation across adjacent denoising timesteps along one trajectory. SPECFOLD instead exploits *cross-branch redundancy* within a speculative verification pass.

2.3 Redundancy Exploitation in Transformer Inference

Redundancy in Transformer inference has been reduced along two axes. The first is the *temporal axis*: reusing activations across successive decoding or denoising steps, as in autoregressive KV caching, cross-request prefix sharing (Kwon et al., 2023; Zheng et al., 2024; Juravsky et al., 2024), and cross-timestep feature caching in diffusion models (Ma et al., 2024, 2025). The second is *within-pass sparsity*: skipping or merging redundant tokens and layers inside a single forward call via token merging and pruning (Bolya et al., 2023; Wang et al., 2021; Rao et al., 2021) or adaptive early exit (Xin et al., 2020; Schuster et al., 2022). CALM (Schuster et al., 2022) is the closest precedent to our approach, using cosine distance between consecutive-layer hidden states as the exit signal. Multi-branch speculative decoding (Miao et al., 2024; Cai et al., 2024; Li et al., 2024) batches a tree of draft continuations into one verification pass and reuses the shared-prefix KV cache across tree nodes, but recomputes every sibling branch's suffix activations independently. SPECFOLD targets a third axis:

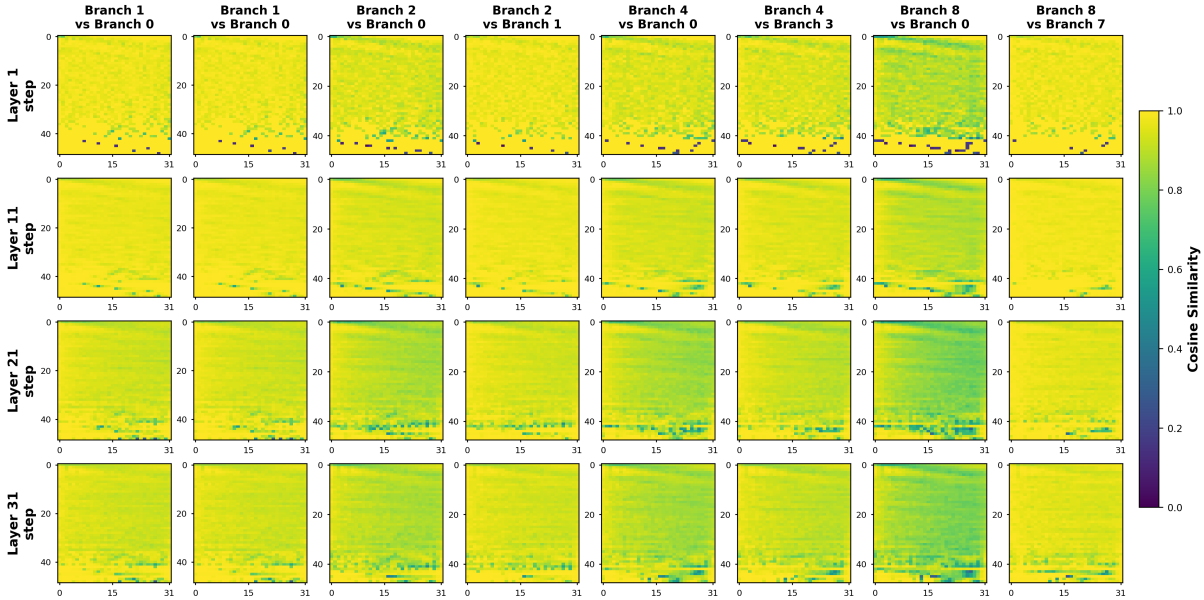


Figure 1: Per-token pre-QKV hidden-state cosine similarity for selected draft branches in one speculative verification pass on Fast-dLLM-v1 LLaDA-8B. Columns compare each draft branch either with the main branch (Branch 0) or with its immediate parent in the draft graph. Rows show representative layers spanning the model depth: layers 1, 11, 21, and 31. Yellow indicates high cosine similarity. Draft branches remain highly similar to their immediate parents across most tokens and layers, with divergence concentrated near recently unmasked positions.

cross-branch redundancy within the verification pass itself, folding sibling draft branches that inherit nearly identical hidden states from a common parent, without modifying the draft-construction policy or model weights.

3 The Proposed SpecFold Framework

SpecFold builds on the algorithmic observation: speculative-decoding verification exhibits strong cross-branch redundancy that dense kernels fail to exploit because the resulting active sets vary across steps. As noted in Sec. 2.2, each draft branch inherits all but up to S_t newly unmasked positions from its parent, suggesting that much of the per-branch computation in Eq. (2) is redundant. We characterize this redundancy (Sec. 3.1), then present the SpecFold algorithm (Sec. 3.2). We extend the notation of Sec. 2.2 with one additional definition: we write $\mathbf{H}^{\ell}_{i,k} \in \mathbb{R}^{N\times d}$ for the pre-QKV hidden state at layer ℓ, denoising block k, and branch i, with $\mathbf{H}^{\ell}_{i,k}[n]$ denoting the hidden state of the n-th token position.

3.1 SpecFold: Profiling the Redundancy

Where redundancy lives. By construction, each draft branch $X_{i,k}$ inherits all but up to S_t newly unmasked positions from its parent $X_{\mathrm{par}(i),k}$. We therefore expect the corresponding pre-QKV hidden states to satisfy $\mathbf{H}^{\ell}_{i,k}[n] \approx \mathbf{H}^{\ell}_{\mathrm{par}(i),k}[n]$ on inherited positions n at every layer ℓ. That is, the per-branch forward pass repeats work that was already done for the parent or the main branch. We now test this prediction empirically.

Empirical profile. We test these predictions on the existence of redundancy on Fast-dLLM-v1 LLaDA-8B (Nie et al., 2025; Wu et al., 2025b) (GSM8K (Cobbe et al., 2021), generation length 256, block size $N = 32$, calibrated draft graph with $B = 8$ drafts; statistics over 100 samples). Fig. 1 visualizes the cosine similarity of $\mathbf{H}^{\ell}_{i,k}[n]$ and $\mathbf{H}^{\ell}_{\mathrm{par}(i),k}[n]$ across drafts (i), layers (ℓ) and token positions (n) within the block. The following observations summarize the corresponding measurements.

Obs. 1 (Hidden states match across parent–child pairs). $>80\%$ of (layer, token, step) triples have large parent-child hidden state cosine similarity across layers 1, 11, 21, 31. Low-similarity regions concentrate at positions recently unmasked by draft i, exactly where the child draft contributes new information. This supports that $\mathbf{H}^{\ell}_{i,k}[n] \approx \mathbf{H}^{\ell}_{\mathrm{par}(i),k}[n]$ on inherited positions n.

Obs. 2 (Parent context is the appropriate reuse reference for draft branches). Draft-vs-parent similarity exceeds draft-vs-main because draft and parent share more unmasked tokens by construction. The right reference for reuse is therefore the parent, not the main branch.

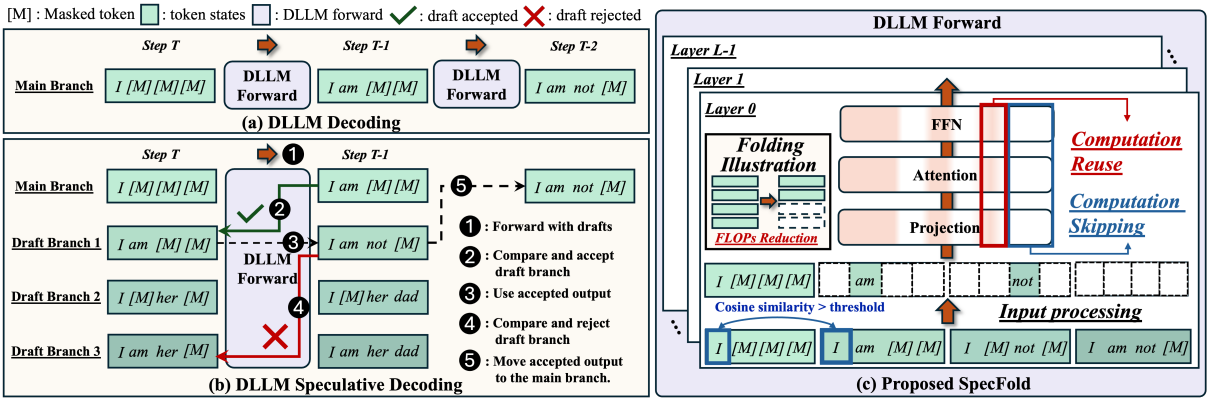


Figure 2: Overview of SPECFOLD. (a) Standard DLLM decoding. A sequence containing masked tokens [M] is processed by the DLLM, which unmasks a subset of tokens at each forward pass, progressively refining the output over multiple timesteps. (b) Speculative decoding for DLLMs. Alongside the main branch, several draft branches are constructed and processed with the main branch in a single batched forward pass at step T. At step $T-1$, each draft is compared against the updated main branch: if a draft matches (Branch 1), its precomputed outputs are reused to advance the main branch directly to step $T-2$, skipping a DLLM forward pass. (c) The proposed SPECFOLD mechanism, which exploits redundancy across draft branches within a single DLLM forward pass. Prior to the QKV projection of each Transformer layer, tokens whose hidden states exceed a cosine-similarity threshold are folded together, and their computation is reused across the projection, attention, and FFN sublayers. This reduces FLOPs in the compute-bound regime.

3.2 SPECFOLD: Folding Redundant Drafts

SPECFOLD exploits the cross-branch redundancy characterized in Sec. 3.1 to reduce the FLOPs of the batched speculative forward pass. The key insight is that, for a parent–child draft pair $(\mathrm{par}(i),\, i)$, the hidden states $\mathbf{H}^{\ell}_{i,k}[n]$ at inherited token positions are nearly identical across all layers (Obs. 1). Rather than recomputing these redundant activations, SPECFOLD identifies them via a per-token parent-child cosine similarity check before the QKV projection at each Transformer layer, and folds the child's computation into the parent's hidden states at those positions.

Formally, at each layer ℓ and denoising block k, for each draft branch $i > 0$ and each token position n, we compute the cosine similarity between $\mathbf{H}^{\ell}_{i,k}[n]$ and $\mathbf{H}^{\ell}_{\mathrm{par}(i),k}[n]$:

$$\sigma^{\ell}_{i,k,n} \;=\; \frac{\langle \mathbf{H}^{\ell}_{i,k}[n],\; \mathbf{H}^{\ell}_{\mathrm{par}(i),k}[n] \rangle}{\|\mathbf{H}^{\ell}_{i,k}[n]\| \; \|\mathbf{H}^{\ell}_{\mathrm{par}(i),k}[n]\|}. \tag{3}$$

If $\sigma^{\ell}_{i,k,n} \geq \tau$, token n of draft i is *folded* at layer ℓ: we set $\mathbf{H}^{\ell}_{i,k}[n] \leftarrow \mathbf{H}^{\ell}_{\mathrm{par}(i),k}[n]$. The resulting reused and recomputed position sets are:

$$\mathcal{R}^{\ell}_{i} \;=\; \{n : \sigma^{\ell}_{i,k,n} \geq \tau\}, \quad \overline{\mathcal{R}}^{\ell}_{i} \;=\; [N] \setminus \mathcal{R}^{\ell}_{i}. \tag{4}$$

Alg. 1 summarizes one complete SPECFOLD step. In Alg. 1, there is a FOLDED procedure that materializes the FLOPs reduction. See Appendix A for the detailed implementation.

Algorithm 1 One SPECFOLD step.

Require: Branches $\{X^{t}_{i,k}\}_{i=0}^{B}$, cosine similarity threshold τ, LM head W_{LM}, number of layers L
Ensure: Output logits $\{p^{t}_{i,k}\}_{i=0}^{B}$ for all branches

Embed all branches: $\mathbf{H}^{0}_{i,k} \leftarrow \mathrm{Embed}(X^{t}_{i,k})$ for all $i \in \{0, \ldots, B\}$
for each layer $\ell = 0, \ldots, L-1$ **do**
  for each draft $i = 1, \ldots, B$ **do**
    Compute $\sigma^{\ell}_{i,k,n}$ via Eq. (3) for all n
    $\mathcal{R}^{\ell}_{i} \leftarrow \{n : \sigma^{\ell}_{i,k,n} \geq \tau\}$, $\overline{\mathcal{R}}^{\ell}_{i} \leftarrow [N] \setminus \mathcal{R}^{\ell}_{i}$ ▷ fold / recompute sets
    $\mathbf{H}^{\ell}_{i,k}[\mathcal{R}^{\ell}_{i}] \leftarrow \mathbf{H}^{\ell}_{\mathrm{par}(i),k}[\mathcal{R}^{\ell}_{i}]$ ▷ replace with parent hidden states
  end for
  $\{\mathbf{H}^{\ell+1}_{i,k}\}_{i=0}^{B} \leftarrow \mathrm{FOLDED}(\{\mathbf{H}^{\ell}_{i,k}\}_{i=0}^{B}, \mathrm{par}(\cdot), \mathcal{R}^{\ell}_{i}, \overline{\mathcal{R}}^{\ell}_{i})$ ▷ Alg. 2, Appendix A
end for
$p^{t}_{i,k} \leftarrow W_{\mathrm{LM}}\, \mathbf{H}^{L}_{i,k}$ for all $i \in \{0, \ldots, B\}$
Cache $\{(X^{t}_{i,k},\, p^{t}_{i,k})\}_{i=0}^{B}$ for speculative verification

4 Experiments

4.1 Setup

Models and baselines. We use Fast-dLLM-v2 as the primary evaluation testbed and evaluate SPECFOLD on Fast-dLLM-v2-1.5B and Fast-dLLM-v2-7B (Wu et al., 2025a), two representative block-diffusion language models. Unless otherwise specified, we follow the original Fast-dLLM-v2 decoding configuration: the generation block size is set to 32, the small-block size to 8, the parallel decoding confidence threshold to 0.9, and block caching is disabled. We further evaluate on Fast-dLLM-

Table 1: **Results of SPECFOLD on Fast-dLLM-v2-7B, Fast-dLLM-v2-1.5B, Fast-dLLM-v1 LLaDA-8B, and Fast-dLLM-v1 Dream-7B across five benchmarks.** We report average NFE, average TFLOPs, and task accuracy. Green NFE annotations denote reductions relative to Vanilla decoding, while green TFLOPs annotations denote reductions of SPECFOLD relative to Spiffy. This separates the NFE reduction achieved by speculative decoding from the additional cross-branch computation saved by SPECFOLD within the speculative verification pass.

Benchmark	Method	Fast-dLLM-v2-7B			Fast-dLLM-v2-1.5B			LLaDA-8B			Dream-7B		
		NFE↓	TFLOPs↓	Acc.↑	NFE↓	TFLOPs↓	Acc.↑	NFE↓	TFLOPs↓	Acc.↑	NFE↓	TFLOPs↓	Acc.↑
GSM8K	Vanilla	113	52	81.43%	113	11	63.07%	86	199	78.01%	161	219	73.69%
	Spiffy	$98_{\downarrow 13\%}$	275	81.58%	$93_{\downarrow 18\%}$	58	61.79%	$62_{\downarrow 28\%}$	294	76.95%	$105_{\downarrow 35\%}$	368	75.13%
	SPECFOLD	$98_{\downarrow 13\%}$	$170_{\downarrow 38\%}$	80.52%	$88_{\downarrow 22\%}$	$30_{\downarrow 48\%}$	61.18%	$63_{\downarrow 27\%}$	$213_{\downarrow 28\%}$	74.91%	$107_{\downarrow 34\%}$	$236_{\downarrow 36\%}$	73.31%
MATH	Vanilla	227	105	58.54%	251	26	37.14%	111	179	36.66%	120	173	38.56%
	Spiffy	$189_{\downarrow 17\%}$	528	58.04%	$206_{\downarrow 18\%}$	128	37.04%	$75_{\downarrow 32\%}$	294	36.82%	$80_{\downarrow 33\%}$	290	38.64%
	SPECFOLD	$189_{\downarrow 17\%}$	$201_{\downarrow 62\%}$	58.43%	$205_{\downarrow 18\%}$	$49_{\downarrow 62\%}$	36.32%	$78_{\downarrow 30\%}$	$196_{\downarrow 33\%}$	36.44%	$84_{\downarrow 30\%}$	$190_{\downarrow 35\%}$	37.60%
HumanEval	Vanilla	156	73	65.85%	161	16	37.50%	90	90	37.80%	153	112	52.44%
	Spiffy	$116_{\downarrow 26\%}$	333	64.02%	$119_{\downarrow 26\%}$	74	35.98%	$62_{\downarrow 31\%}$	182	37.20%	$100_{\downarrow 35\%}$	251	50.61%
	SPECFOLD	$104_{\downarrow 33\%}$	$149_{\downarrow 55\%}$	63.37%	$113_{\downarrow 30\%}$	$42_{\downarrow 43\%}$	35.06%	$64_{\downarrow 29\%}$	$102_{\downarrow 44\%}$	35.98%	$99_{\downarrow 35\%}$	$125_{\downarrow 50\%}$	48.78%
MBPP	Vanilla	163	76	62.30%	153	16	45.90%	71	156	37.80%	111	158	53.00%
	Spiffy	$115_{\downarrow 29\%}$	334	63.49%	$120_{\downarrow 22\%}$	76	44.58%	$49_{\downarrow 31\%}$	228	38.40%	$75_{\downarrow 32\%}$	266	52.60%
	SPECFOLD	$115_{\downarrow 29\%}$	$163_{\downarrow 51\%}$	63.58%	$125_{\downarrow 18\%}$	$46_{\downarrow 39\%}$	45.37%	$49_{\downarrow 31\%}$	$173_{\downarrow 24\%}$	36.80%	$77_{\downarrow 31\%}$	$187_{\downarrow 30\%}$	50.80%
IFEval	Vanilla	270	123	64.69%	294	30	48.24%	137	101	53.79%	184	117	34.75%
	Spiffy	$207_{\downarrow 23\%}$	625	64.51%	$224_{\downarrow 24\%}$	152	47.50%	$91_{\downarrow 34\%}$	238	54.53%	$122_{\downarrow 34\%}$	293	34.20%
	SPECFOLD	$186_{\downarrow 31\%}$	$323_{\downarrow 48\%}$	63.96%	$213_{\downarrow 28\%}$	$79_{\downarrow 48\%}$	46.77%	$86_{\downarrow 37\%}$	$187_{\downarrow 21\%}$	52.68%	$120_{\downarrow 35\%}$	$135_{\downarrow 54\%}$	34.57%

v1, covering LLaDA-8B and Dream-7B, to assess generalization across DLLM families. For these models, we set the generation length to 256 and the block size to 32, use parallel decoding with a confidence threshold of 0.9, and enable the dual-cache mechanism following Wu et al. (2025b). Across all models, Spiffy constructs a draft graph, and SPECFOLD reuses the same draft graph with a similarity threshold of $\tau = 0.96$. We compare the following decoding strategies:

- *Vanilla decoding.* The native block-wise decoding of each model, without speculative drafts.
- *Spiffy* (Agrawal et al., 2025). A calibration-based speculative decoding baseline. For each model–dataset pair, we use 50 calibration samples to construct a degree-1 draft graph, and instantiate B=8 draft branches at inference time according to the calibrated token-position and vocabulary-rank patterns.
- SPECFOLD. Our sparsity-aware speculative decoding method. SPECFOLD uses the same draft graph construction algorithm as Spiffy, but avoids redundant computation across related draft branches by reusing parent draft computations when the token-level representation similarity $\sigma^{\ell}_{i,k,n}$ (Eq. 3) exceeds a threshold τ.

Benchmarks and Metrics. Our evaluation spans three domains: mathematical reasoning (GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021)), code generation (HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021)), and instruction following (IFEval (Zhou et al., 2023)). Following Fast-dLLM-v2's evaluation setup, we use zero-shot prompting for all tasks. Non-code benchmarks are evaluated with LM-Eval Harness, and HumanEval and MBPP are evaluated with EvalPlus (Liu et al., 2023). We report three primary metrics: Number of Function Evaluations (NFE) per sample, TFLOPs per sample, and task-standard accuracy. NFE measures the number of denoising steps required for generation, while TFLOPs quantify computational cost. All experiments are run on a single NVIDIA H100 80GB GPU.

4.2 Main Results

Tab. 1 presents our primary results across all four models. For each model and benchmark, we compare decoding efficiency and task accuracy using average Number of Function Evaluations (NFE), average TFLOPs, and the task-standard accuracy metric. Our evaluation separates two efficiency effects. NFE reductions are reported relative to Vanilla decoding, since speculative decoding is designed to skip future denoising calls. TFLOPs reductions, however, are reported relative to Spiffy, since SPECFOLD specifically targets the dense multi-branch verification overhead introduced by speculative decoding. Note that SPECFOLD does not aim to reduce total FLOPs below Vanilla.

Fast-dLLM-v2. SPECFOLD consistently reduces speculative-verification cost across both model scales. For Fast-dLLM-v2-7B, SPECFOLD reduces NFE by 13–33% relative to Vanilla decoding and reduces TFLOPs by 38–62% relative to Spiffy across the five benchmarks. For Fast-dLLM-

v2-1.5B, SPECFOLD reduces NFE by 18–30% relative to Vanilla and reduces TFLOPs by 39–62% relative to Spiffy. These results show that cross-branch folding removes a substantial fraction of the additional computation introduced by speculative verification.

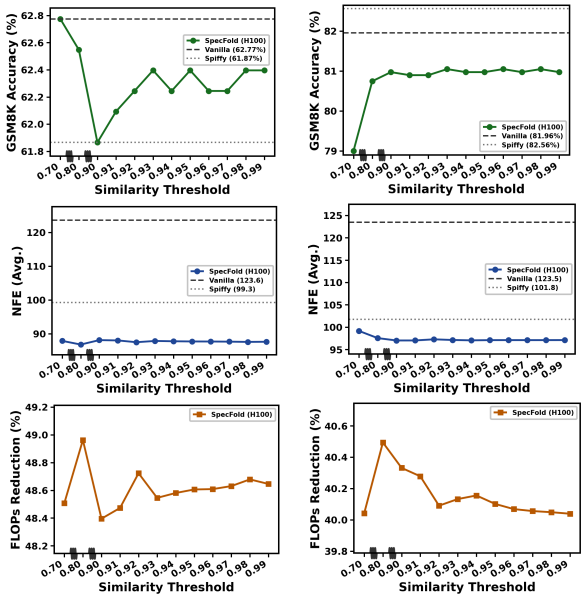
Accuracy remains competitive under this efficiency-oriented operating point. On Fast-dLLM-v2-1.5B, SPECFOLD stays within roughly one percentage point of Spiffy across all five benchmarks, while achieving sizable TFLOPs reductions. On Fast-dLLM-v2-7B, SPECFOLD preserves accuracy closely on GSM8K, MATH, and IFEval, while code-generation benchmarks exhibit a larger accuracy–efficiency trade-off. This suggests that SPECFOLD is most effective when parent–child similarity remains high enough to support aggressive folding, and that task-dependent threshold calibration may further improve the trade-off on code-generation tasks.

LLaDA-8B and Dream-7B. SPECFOLD consistently reduces TFLOPs relative to Spiffy across both models while maintaining competitive task accuracy. On LLaDA-8B, TFLOPs reductions relative to Spiffy range from 21% to 44% across the five benchmarks. On Dream-7B, the corresponding reductions range from 30% to 54%. NFE reductions relative to Vanilla decoding are similarly consistent, ranging from 27%–37% on LLaDA-8B and 30%–35% on Dream-7B. These results confirm that SPECFOLD is not tied to Fast-dLLM-v2-specific implementation details and that the folding principle transfers to other block-wise diffusion language models.

4.3 Ablations

Threshold sweep. We evaluate the sensitivity of SPECFOLD to the similarity threshold τ on GSM8K with Fast-dLLM-v2-1.5B and 7B on H100. We sweep $\tau \in \{0.70, .., 0.99\}$ while keeping all other decoding hyperparameters fixed. Fig. 3 reports accuracy, average NFE, and FLOPs reduction across thresholds, with Vanilla and Spiffy shown as horizontal baselines. Deployment thresholds are selected at the speedup–accuracy knee under a small allowed accuracy drop.

Across both Fast-dLLM-v2-1.5B and 7B, SPECFOLD consistently reduces NFE relative to Vanilla and remains below the Spiffy baseline across the sweep. The FLOPs reduction is also stable, staying close to 50% for the 1.5B model and around 40%



(a) Fast-dLLM-v2-1.5B. (b) Fast-dLLM-v2-7B.

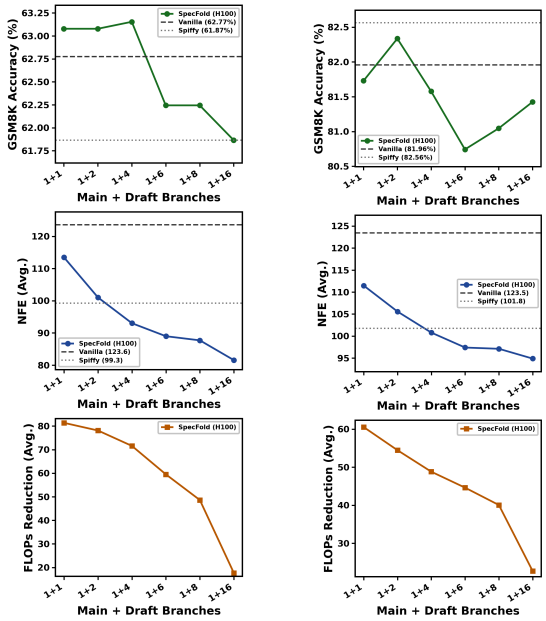
Figure 3: Similarity-threshold sensitivity analysis of SPECFOLD on GSM8K. For each model, we report task accuracy, average NFE, and FLOPs reduction across different similarity thresholds. Dashed and dotted horizontal lines indicate the Vanilla and Spiffy baselines, respectively. Rug markers on the x-axis show the distribution of selected similarity thresholds.

for the 7B model over most thresholds. Accuracy is more sensitive in the low-threshold regime, where aggressive reuse may accept less reliable speculative branches. However, once τ is set to a moderate or high value, accuracy becomes stable while most of the NFE and FLOPs benefits are preserved.

These results indicate that SPECFOLD has a broad and robust operating region, and its efficiency gains do not depend on fine-tuning τ precisely. We therefore fix $\tau = 0.96$ at the knee across all models and benchmarks without per-task tuning, and leave task-adaptive thresholds to future work.

Number of draft branches sweep. We analyze the sensitivity of SPECFOLD to the number of draft branches in speculative decoding. On GSM8K with Fast-dLLM-v2-1.5B and 7B on H100, we sweep the main-plus-draft branch configuration over $\{1{+}1, 1{+}2, 1{+}4, 1{+}6, 1{+}8, 1{+}16\}$, while keeping all other hyperparameters fixed. Fig. 4 reports accuracy, average NFE, and FLOPs reduction, with Vanilla and Spiffy shown as horizontal baselines.

Across both model sizes, increasing the number of draft branches consistently reduces average NFE, as more speculative candidates can be explored and accepted per denoising step. However, the marginal NFE gain saturates at larger branch counts, while



(a) Fast-dLLM-v2-1.5B. (b) Fast-dLLM-v2-7B.

Figure 4: GSM8K draft-branch sensitivity of SPECFOLD for Fast-dLLM-v2-1.5B and Fast-dLLM-v2-7B. For each model, we report task accuracy, average NFE, and FLOPs reduction under different main-plus-draft configurations. Dashed and dotted horizontal lines indicate the Vanilla and Spiffy baselines, respectively.

FLOPs reduction over Spiffy decreases because additional branches introduce more speculative computation. Accuracy is stable for moderate branch counts but becomes more variable under aggressive branching. These results suggest that moderate draft-branch settings offer the best operating region, capturing most of the NFE reduction while preserving substantial FLOPs savings and competitive accuracy.

4.4 Analysis of Similarity Ripple

To test whether the pre-QKV hidden state $\mathbf{H}^{\ell}_{i,k}[n]$ provides a reliable basis for the folding decision in Eq. (3), we compare a parent-child $(\mathrm{par}(i), i)$ pair that differ by only one unmasked token ("edited position") in the speculative draft branch. Fig. 5 shows that the resulting change in cosine similarity $\sigma^{\ell}_{i,k,n}$ is structured rather than arbitrary. The edited position creates a persistent low-similarity band, but many other positions retain high similarity for a large fraction of layers. In other words, a local edit does propagate through self-attention, yet it does not immediately invalidate the hidden state $\mathbf{H}^{\ell}_{i,k}[n]$ at every layer–position in the draft branch.

Propagation varies by model. Fast-dLLM-v1 LLaDA-8B remains stable in early layers before similarity drops in deeper layers, while Fast-dLLM-v1 Dream exhibits stronger ripple effects with broader low-similarity regions. Fast-dLLM-v2 shows the highest stability, maintaining high similarity away from edited positions across all layers. These differences highlight that a fixed reuse policy would be overly conservative for stable models and too aggressive where edits propagate more strongly.

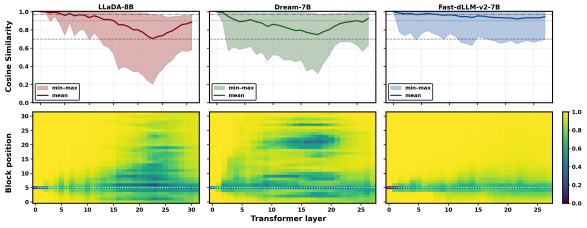


Figure 5: **Per-position pre-QKV hidden state cosine similarity between two inputs with a single token unmasked**. The models are prompted with a GSM8K sample. The parent draft is fully masked, while the child draft inherits all positions except that position $n = 5$ is unmasked. We measure σ^{ℓ}_{n} as defined in Eqn. 3. **Top row**: mean similarity over positions other than the edited one, with shaded min–max range. Grey dashed lines mark candidate reuse thresholds of 0.70, 0.90, and 0.96. **Bottom row**: full per-position similarity heatmaps across layers. The white dotted line marks the block position where the parent and child drafts differ. Results are shown for Fast-dLLM-v1 LLaDA-8B, Fast-dLLM-v1 Dream-7B, and Fast-dLLM-v2-7B.

This observation motivates the per-token similarity gate in Eq. (3). Because $\sigma^{\ell}_{i,k,n}$ is measured immediately before the QKV projection, the gate can decide, at each layer ℓ and position n, whether $\mathbf{H}^{\ell}_{\mathrm{par}(i),k}[n]$ remains representative of $\mathbf{H}^{\ell}_{i,k}[n]$ before redoing the expensive attention and FFN. Positions in $\mathcal{R}^{\ell}_{i}$ provide direct reuse opportunities, while the low-similarity band identifies recomputing positions in $\overline{\mathcal{R}}^{\ell}_{i}$. The gate therefore matches the empirical granularity of the ripple: it preserves reuse where $\mathbf{H}^{\ell}_{i,k}[n] \approx \mathbf{H}^{\ell}_{\mathrm{par}(i),k}[n]$, but falls back to fresh computation where the child has diverged.

5 Conclusion

This paper targets a key bottleneck in speculative diffusion decoding: the dense verification pass over highly similar draft branches. We show that this pass contains substantial *cross-branch activation redundancy*, which SPECFOLD exploits through branch folding during verification. Across four DLLMs and five benchmarks, SPECFOLD improves throughput over Spiffy while preserving task accuracy, demonstrating consistent efficiency gains.

6 Limitations

Approximation-induced accuracy variation. SPECFOLD reuses parent draft computations at folded positions and is therefore approximate at those positions. Although the similarity threshold τ controls this approximation and our sensitivity analysis shows a broad stable operating region, overly aggressive reuse can still introduce accuracy fluctuations on some benchmarks.

Dependence on speculative branch structure. The achievable FLOPs reduction depends on the number of draft branches and the amount of redundancy among them. When draft branches become too diverse, fewer positions can be folded safely, reducing the benefit of SPECFOLD.

Model and task coverage. Our evaluation covers four DLLMs across five benchmarks. While these results suggest that SPECFOLD applies broadly across block-wise diffusion language models, evaluation across additional architectures, decoding schedules, and long-context settings remains future work.

FLOPs reduction as latency improvement. SPECFOLD reports FLOPs, the architecture-agnostic measure of speculative verification cost. Translating those FLOPs into latency improvement requires a fused on-chip kernel for the folded layer, analogous to FlashAttention (Dao et al., 2022) relative to standard attention. We provide a roofline analysis of when folding is latency-profitable in Appendix C and leave the fused-kernel implementation to future systems work.

References

Sudhanshu Agrawal, Risheek Garrepalli, Raghavv Goel, Mingu Lee, Christopher Lott, and Fatih Porikli. 2025. Spiffy: Multiplying diffusion LLM acceleration via lossless speculative decoding. *arXiv preprint arXiv:2509.18085*.

Marianne Arriola, Aaron Gokaslan, Justin T. Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sekhar Sahoo, and Volodymyr Kuleshov. 2025. Block diffusion: Interpolating between autoregressive and diffusion language models. *arXiv preprint arXiv:2503.09573*.

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

Tiwei Bie, Maosong Cao, Xiang Cao, Bingsen Chen, Fuyuan Chen, Kun Chen, Lun Du, Daozhuo Feng, Haibo Feng, Mingliang Gong, and 1 others. 2026. LLaDA2.1: Speeding up text diffusion via token editing. *arXiv preprint arXiv:2602.08676*.

Tiwei Bie, Maosong Cao, Kun Chen, Lun Du, Mingliang Gong, Zhuochen Gong, Yanmei Gu, Jiaqi Hu, Zenan Huang, Zhenzhong Lan, and 1 others. 2025. LLaDA2.0: Scaling up diffusion language models to 100b. *arXiv preprint arXiv:2512.15745*.

Daniel Bolya, Cheng-Yang Fu, Xiaoliang Dai, Peizhao Zhang, Christoph Feichtenhofer, and Judy Hoffman. 2023. Token merging: Your ViT but faster. In *International Conference on Learning Representations*.

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

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

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

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359.

Yifeng Gao, Ziang Ji, Yuxuan Wang, Biqing Qi, Hanlin Xu, and Linfeng Zhang. 2025. Self speculative decoding for diffusion large language models. *arXiv preprint arXiv:2510.04147*.

Raghavv Goel, Risheek Garrepalli, Sudhanshu Agrawal, Chris Lott, Mingu Lee, and Fatih Porikli. 2026. Skip to the good part: Representation structure and inference-time layer skipping in diffusion vs. autoregressive LLMs. *arXiv preprint arXiv:2603.07475*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset. In *Advances in Neural Information Processing Systems*, volume 34.

Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. 2024. Hydragen: High-throughput LLM inference with shared prefixes. *arXiv preprint arXiv:2402.05099*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, pages 611–626.

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

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

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. EAGLE: Speculative sampling requires rethinking feature uncertainty. *Preprint*, arXiv:2401.15077.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*.

Zhiyuan Liu, Yicun Yang, Yaojie Zhang, Junjie Chen, Chang Zou, Qingyuan Wei, Shaobo Wang, and Linfeng Zhang. 2025. dLLM-Cache: Accelerating diffusion large language models with adaptive caching. *arXiv preprint arXiv:2506.06295*.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2024. DeepCache: Accelerating diffusion models for free. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15762–15772.

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

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, and 1 others. 2024. SpecInfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 932–949.

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

Yongming Rao, Wenliang Zhao, Benlin Liu, Jiwen Lu, Jie Zhou, and Cho-Jui Hsieh. 2021. DynamicViT: Efficient vision transformers with dynamic token sparsification. In *Advances in Neural Information Processing Systems*, volume 34, pages 13937–13949.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Tran, Yi Tay, and Donald Metzler. 2022. Confident adaptive language modeling. In *Advances in Neural Information Processing Systems*, volume 35, pages 17456–17472.

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

Wenhao Sun, Rong-Cheng Tu, Yifu Ding, Zhao Jin, Jingyi Liao, Yongcheng Jing, and Dacheng Tao. 2026. SPA-Cache: Singular proxies for adaptive caching in diffusion language models. *Preprint*, arXiv:2602.02544.

Hanrui Wang, Zhekai Zhang, and Song Han. 2021. SpAtten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture*, pages 97–110.

Chengyue Wu, Hao Zhang, Shuchen Xue, Shizhe Diao, Yonggan Fu, Zhijian Liu, Pavlo Molchanov, Ping Luo, Song Han, and Enze Xie. 2025a. Fast-dLLM v2: Efficient block-diffusion LLM. *arXiv preprint arXiv:2509.26328*.

Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. 2025b. Fast-dLLM: Training-free acceleration of diffusion LLM by enabling KV cache and parallel decoding. *arXiv preprint arXiv:2505.22618*.

Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. 2020. DeeBERT: Dynamic early exiting for accelerating BERT inference. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2246–2251.

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

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. 2023. Instruction-following evaluation for large language models. *Preprint*, arXiv:2311.07911.

Fengqi Zhu, Zebin You, Yipeng Xing, Zenan Huang, Lin Liu, Yihong Zhuang, Guoshan Lu, Kangyu Wang, Xudong Wang, Lanning Wei, Hongrui Guo, Jiaqi Hu, Wentao Ye, Tieyuan Chen, Chenchen Li, Chengfu Tang, Haibo Feng, Jun Hu, Jun Zhou, and 7 others. 2025. LLaDA-MoE: A sparse MoE diffusion language model. *Preprint*, arXiv:2509.24389.

Appendix

- Sec. A: Folded Transformer Algorithm
  - Folded Self-attention
  - Folded FFN
  - Putting it Together: One Folded Layer
- Sec. B: On the Numerical Discrepancy with Vanilla Decoding
- Sec. C: Roofline Analysis of Folded Computation
  - Roofline regime
  - Empirical confirmation
  - Two operating regimes for SPECFOLD

A Folded Transformer Algorithm

This appendix develops the per-layer computation of SPECFOLD in detail. Given the folded position set $\mathcal{R}_i^\ell$ and recomputed set $\overline{\mathcal{R}}_i^\ell$ from Eq. (4), we describe how each Transformer sublayer is restructured to avoid redundant computation. For recomputed positions $\overline{\mathcal{R}}_i$, the computation is *mathematically equivalent* to the dense forward pass. For folded positions $\mathcal{R}_i$, both the attention and FFN outputs are reused from the parent, introducing approximations whose quality is controlled by the threshold τ and characterized empirically in Sec. 3.1.

For brevity we drop the block index k and layer index ℓ within a single layer's derivation, writing $\mathbf{H}_i$ for $\mathbf{H}_{i,k}^\ell$, p for $\mathrm{par}(i)$, $\mathcal{R}_i$ for $\mathcal{R}_i^\ell$, and $\overline{\mathcal{R}}_i$ for $\overline{\mathcal{R}}_i^\ell$. All $B{+}1$ branches traverse the layer together in a single batched step, so the parent's intermediates are not available in advance — they are produced *within* the same layer call. The folded layer proceeds in four stages:

1. **Parent attention.** Every parent draft runs the standard dense attention, producing O^p and the post-attention residual $\mathbf{H}'_p$. These are kept on-chip.
2. **Draft attention.** Every draft copies O_m^p at folded query rows $m \in \mathcal{R}_i$ and runs dense attention at recomputed query rows $m \in \overline{\mathcal{R}}_i$, assembling $\mathbf{H}'_i$ for all positions.
3. **Parent FFN.** Every parent draft computes $\mathrm{FFN}(\mathbf{H}'_p)$ and adds the residual to obtain $\mathbf{H}_p^+$.
4. **Draft FFN.** Every draft copies $\mathbf{H}_p^+[n]$ at positions $n \in \mathcal{R}_i$ and recomputes $\mathrm{FFN}(\mathbf{H}'_i[n])$ at positions $n \in \overline{\mathcal{R}}_i$.

The main branch X^0 has no parent and participates only in the parent stages (1 and 3). A branch that is both a draft of some parent and a parent of further drafts is processed in the draft stages with respect to its own parent, and its outputs are then used by descendants.

A.1 Folded Self-attention

Self-attention is not position-wise: the output at row m depends on every key and value in the sequence. SPECFOLD handles folded and recomputed query rows differently based on the pre-attention partition $(\mathcal{R}_i, \overline{\mathcal{R}}_i)$.

Setup. Let $Q^i, K^i, V^i \in \mathbb{R}^{N\times d_h}$ be the per-head projections for draft i, and analogously Q^p, K^p, V^p for the parent. Define $S_{mn}^i = q_m^i \cdot k_n^i/\sqrt{d_h}$, with softmax denominator $Z_m^i = \sum_n \exp(S_{mn}^i)$, unnormalized output $\tilde{O}_m^i = \sum_n \exp(S_{mn}^i)\, v_n^i$, and normalized output $O_m^i = \tilde{O}_m^i/Z_m^i$. The parent attention stage produces $(Q^p, K^p, V^p, \tilde{O}^p, Z^p, O^p)$, kept on-chip throughout Stage 2.

What folding implies for Q, K, V. Because the QKV projection is linear and per-token, $\mathbf{H}_i[n] = \mathbf{H}_p[n]$ for $n \in \mathcal{R}_i$ implies

$$q_n^i = q_n^p, \quad k_n^i = k_n^p, \quad v_n^i = v_n^p \qquad \forall\, n \in \mathcal{R}_i. \tag{5}$$

Thus Q^i, K^i, and V^i differ from Q^p, K^p, and V^p only on rows in $\overline{\mathcal{R}}_i$, which are computed fresh. The folded rows are inherited from the parent's on-chip QKV buffers.

There are two cases for the query row.

Case A: $m \in \mathcal{R}_i$ folded query. Since $q_m^i = q_m^p$ by Eq. (5), define the unnormalized attention scores

$$\begin{aligned} a_{mn}^p &\triangleq \exp(q_m^p \cdot k_n^p/\sqrt{d_h}), \\ a_{mn}^i &\triangleq \exp(q_m^p \cdot k_n^i/\sqrt{d_h}). \end{aligned} \tag{6}$$

Then the exact attention output for draft i at this row is

$$O_m^i = \frac{\sum_{n\in\mathcal{R}_i} a_{mn}^p v_n^p + \sum_{n\in\overline{\mathcal{R}}_i} a_{mn}^i v_n^i}{\sum_{n\in\mathcal{R}_i} a_{mn}^p + \sum_{n\in\overline{\mathcal{R}}_i} a_{mn}^i}. \tag{7}$$

Here the first sum in both numerator and denominator runs over keys and values shared with the parent, while the second sum runs over draft-specific keys and values at recomputed positions. The parent's attention output is

$$O_m^p = \frac{\sum_{n=0}^{L-1} a_{mn}^p v_n^p}{\sum_{n=0}^{L-1} a_{mn}^p}. \tag{8}$$

Comparing Eq. (7) and Eq. (8), O_m^i and O_m^p share the same query q_m^p and agree on all keys and values at $\mathcal{R}_i$, but differ at the $|\overline{\mathcal{R}}_i|$ positions where the draft has fresh keys and values. SPECFOLD replaces the exact computation with the parent copy:

$$O_m^i \leftarrow O_m^p \qquad \forall\, m \in \mathcal{R}_i, \tag{9}$$

which costs zero arithmetic because O_m^p is already on-chip. The approximation error of this substitution is controlled by the key/value divergence at $\overline{\mathcal{R}}_i$: when $|\overline{\mathcal{R}}_i|$ is small or $(k_n^i, v_n^i) \approx (k_n^p, v_n^p)$ for $n \in \overline{\mathcal{R}}_i$, the two attention outputs remain close.

Case B: $m \in \overline{\mathcal{R}}_i$ recomputed query. Here $q_m^i \neq q_m^p$ in general, so the parent's row provides no direct shortcut. We assemble K^i and V^i by inheriting rows in $\mathcal{R}_i$ from the parent's on-chip QKV buffers and using freshly computed rows in $\overline{\mathcal{R}}_i$, then compute O_m^i via standard dense attention at cost $O(Nd_h)$ per query row.

After both cases, the post-attention residual is formed for all positions:

$$\mathbf{H}_i'[n] = \mathbf{H}_i[n] + W_O O_n^i \qquad \forall\, n \in \{0, \ldots, N-1\}. \tag{10}$$

For $n \in \mathcal{R}_i$, this gives $\mathbf{H}_i'[n] = \mathbf{H}_p[n] + W_O O_n^p = \mathbf{H}_p'[n]$ up to the same approximation as Eq. (9), since both the hidden state and the attention output are copied from the parent.

A.2 Folded FFN

The FFN sublayer is position-wise: $\mathrm{FFN}(\mathbf{H}_i')[n]$ depends only on $\mathbf{H}_i'[n]$. For positions $n \in \mathcal{R}_i$, both $\mathbf{H}_i[n] = \mathbf{H}_p[n]$ and $O_n^i \leftarrow O_n^p$ (Eq. (9)), so the post-attention residual satisfies $\mathbf{H}_i'[n] = \mathbf{H}_p'[n]$ under the same approximation, and therefore

$$\mathrm{FFN}(\mathbf{H}_i')[n] = \mathrm{FFN}(\mathbf{H}_p')[n] \qquad \forall\, n \in \mathcal{R}_i \tag{11}$$

to the same order of approximation. The folded FFN copies the parent's full post-layer output $\mathbf{H}_p^+[n]$ at reuse positions and recomputes at $\overline{\mathcal{R}}_i$:

$$\mathbf{H}_i^+[n] = \begin{cases} \mathbf{H}_p^+[n], & n \in \mathcal{R}_i, \\ \mathbf{H}_i'[n] + \mathrm{FFN}(\mathbf{H}_i'[n]), & n \in \overline{\mathcal{R}}_i. \end{cases} \tag{12}$$

The FLOP count drops from $O(Ld_{\text{ffn}})$ to $O(|\overline{\mathcal{R}}_i| d_{\text{ffn}})$ per branch. The parent's output $\mathbf{H}_p^+$ is produced in Stage 3 and consumed in the same kernel by the draft FFN wave, so the copy is a pure on-chip operation that fuses with the residual add and incurs no HBM traffic.

Remark: post-attention gating as a design alternative. An alternative design inserts a second similarity check between Stages 2 and 3: after assembling $\mathbf{H}_i'$, one compares $\mathbf{H}_i'[n]$ against $\mathbf{H}_p'[n]$ position-by-position with a threshold τ' to obtain a fresh FFN reuse set $\mathcal{F}_i \subseteq \{0, \ldots, L-1\}$ that may differ from $\mathcal{R}_i$. Positions in $\mathcal{R}_i$ where the attention approximation in Eq. (9) introduces noticeable drift would be demoted to FFN recomputation, while positions in $\overline{\mathcal{R}}_i$ whose recomputed attention output remains close to the parent's could be promoted to FFN reuse. This decouples the FFN fold rate from the pre-attention fold rate and grounds each FFN copy in a direct post-attention measurement rather than a pre-layer decision. The additional gate costs $O(Ld)$ per layer per draft, which is negligible relative to the FFN savings. We leave the exploration of this two-threshold design to future work.

A.3 Putting it Together: One Folded Layer

Alg. 2 composes the folded attention and folded FFN into a single batched layer step.

FLOP and bandwidth accounting. Let $r_i = |\overline{\mathcal{R}}_i|/N$ be the fraction of recomputed positions for draft i. Per layer, the dense baseline costs

$$C_{\text{dense}} = c_{\text{qkv}} L d^2 + c_{\text{attn}} L^2 d_h n_h + c_{\text{ffn}} L d\, d_{\text{ffn}}, \tag{13}$$

where n_h is the head count and c_* are constants. Since the folded layer recomputes only $r_i L$ query/position rows, its cost is

$$\begin{aligned} C_{\text{fold}}(r_i) &= r_i c_{\text{qkv}} L d^2 \\ &\quad + r_i c_{\text{attn}} L^2 d_h n_h \\ &\quad + r_i c_{\text{ffn}} L d\, d_{\text{ffn}} \\ &= r_i C_{\text{dense}}. \end{aligned} \tag{14}$$

Thus QKV, attention, and FFN all scale linearly with r_i: folded positions contribute zero arithmetic across all three sublayers, since their attention and FFN outputs are read directly from the parent's on-chip buffers.

Discussions. SPECFOLD introduces approximations only at folded positions $\mathcal{R}_i$. At these positions, the attention output is set to O_m^p instead of the

Algorithm 2 One folded Transformer layer, batched over all $B{+}1$ branches.

procedure FOLDED($\{\mathbf{H}_i\}_{i=0}^{B}$, $\mathrm{par}(\cdot)$, $\{\mathcal{R}_i\}$, $\{\overline{\mathcal{R}}_i\}$)
Stage 1: Parent attention
  for each draft branch j that is a parent, in parallel **do**
    $Q^j, K^j, V^j \leftarrow$ QKV projection of $\mathbf{H}_j$
    $O^j \leftarrow \text{flash-attention}(Q^j, K^j, V^j)$ ▷ kept on-chip
    $\mathbf{H}'_j \leftarrow \mathbf{H}_j + W_O O^j$
  end for
  // Sync: parent outputs O^p and $\mathbf{H}'_p$ are visible
Stage 2: Draft attention
  for each draft i with $p = \mathrm{par}(i)$, in parallel **do**
    Compute $Q^i[\overline{\mathcal{R}}_i]$, $K^i[\overline{\mathcal{R}}_i]$, $V^i[\overline{\mathcal{R}}_i]$ from $\mathbf{H}_i[\overline{\mathcal{R}}_i]$
    $K^i[\mathcal{R}_i] \leftarrow K^p[\mathcal{R}_i]$; $V^i[\mathcal{R}_i] \leftarrow V^p[\mathcal{R}_i]$ ▷ inherit folded K and V on-chip
    for each query row $m \in \mathcal{R}_i$ **do**
      $O^i_m \leftarrow O^p_m$ ▷ copy; see Eq. (9)
    end for
    for each query row $m \in \overline{\mathcal{R}}_i$ **do**
      $O^i_m \leftarrow \text{attention_row}(Q^i_m, K^i, V^i)$ ▷ dense recompute
    end for
    $\mathbf{H}'_i \leftarrow \mathbf{H}_i + W_O O^i$
  end for
  // Sync: all branches have $\mathbf{H}'$
Stage 3: Parent FFN
  for each branch j that is a parent, in parallel **do**
    $\mathbf{H}^+_j \leftarrow \mathbf{H}'_j + \mathrm{FFN}(\mathbf{H}'_j)$ ▷ kept on-chip
  end for
  // Sync: parent output $\mathbf{H}^+_p$ is visible
Stage 4: Draft FFN
  for each draft i with $p = \mathrm{par}(i)$, in parallel **do**
    for each position $n \in \mathcal{R}_i$ **do**
      $\mathbf{H}^+_i[n] \leftarrow \mathbf{H}^+_p[n]$ ▷ copy; see Eq. (12)
    end for
    for each position $n \in \overline{\mathcal{R}}_i$ **do**
      $\mathbf{H}^+_i[n] \leftarrow \mathbf{H}'_i[n] + \mathrm{FFN}(\mathbf{H}'_i[n])$ ▷ recompute
    end for
  end for
  return $\{\mathbf{H}^+_i\}_{i=0}^{B}$
end procedure

exact expression in Eq. (7); the discrepancy is determined by how much the draft's keys and values at $\overline{\mathcal{R}}_i$ differ from the parent's. The FFN copy then follows from the same approximation: because both the hidden state and the attention output at $n \in \mathcal{R}_i$ are taken from the parent, the post-attention residual $\mathbf{H}'_i[n]$ equals $\mathbf{H}'_p[n]$ under the same error, and the position-wise FFN inherits that error directly. The two copies are therefore governed by a single source of approximation, controlled by τ: a tighter τ admits only positions where the draft is very close to the parent, making the key/value divergence at $\overline{\mathcal{R}}_i$ small and all copies accurate. Sec. 3.1 characterizes this quality–efficiency tradeoff empirically.

B On the Numerical Discrepancy with Vanilla Decoding

In principle, speculative decoding is mathematically equivalent to vanilla autoregressive decoding: every accepted draft token is, by construction, the same token vanilla decoding would have produced from the same context (Sec. 2.2). The Spiffy (Agrawal et al., 2025) row in our metric tables should therefore match the Vanilla baseline exactly. In practice, we observe small discrepancies across benchmarks. This appendix explains the source of this discrepancy and describes the sanity check we ran to confirm that our DLLM speculative decoding implementation is correct.

Source of the discrepancy. The gap is numerical rather than algorithmic. As also documented by Spiffy (Agrawal et al., 2025), `torch.bfloat16` matrix multiplication on CUDA GPUs can use an accumulation order that depends on the shape of the input tensor. When speculative decoding batches the main branch together with B draft branches, the matmul reduction is performed over a tensor whose leading dimension differs from the Vanilla case, where no speculative draft is present ($B{=}0$). Even though the draft branches are independent and contribute no information to the main branch's logits in exact arithmetic, the different accumulation order produces rounding differences in the last few bits of each matmul output. These differences are amplified by the short mantissa field of `bfloat16`. Over many denoising steps, they can occasionally flip a token at a position where two candidates have nearly equal logits. Since exact match and pass@1 are non-smooth metrics, a single flipped token can change a sample-level score, yielding the small accuracy gap we observe.

Sanity check: serialized drafts match Vanilla. To confirm that the discrepancy is caused by batched matmul numerics rather than an implementation bug, we ran a sanity check in which the $B{+}1$ branches are forwarded through the DLLM one at a time rather than as a single batched pass. With $B{=}0$ for every call, the matmul shapes match the Vanilla forward exactly, so the accumulation order is preserved. In this serialized mode, our implementation reproduces the Vanilla baseline metrics on every benchmark we tested. This rules out an algorithmic error in the speculative verification logic.

C Roofline Analysis of Folded Computation

We report FLOPs in the main text since SPECFOLD is an algorithmic primitive whose latency realisation depends on kernel-level fusion. This appendix specifies the regime in which folding improves latency, empirically validates that regime, and points to the kernel patterns that close the gap.

C.1 Roofline regime

Let $N_r \triangleq |\overline{\mathcal{R}}_i^\ell| = r_i N$ be the recomputed-row count for draft i, where N is the block length (Sec. 2) and $r_i = |\overline{\mathcal{R}}_i^\ell|/N$ the recompute fraction (Appendix A.3). For a folded GEMM (the same analysis applies to the self-attention kernel) with N_r recomputed rows and generic weight dimensions $d_{\text{in}}, d_{\text{out}}$ in bf16 (e.g. $d_{\text{in}}{=}d$, $d_{\text{out}}{=}d_{\text{ffn}}$ for an FFN projection), the per-projection arithmetic intensity in the weight-dominated regime ($N_r \ll d_{\text{in}}, d_{\text{out}}$) satisfies

$$\mathcal{I}(N_r) \approx N_r \ \text{ FLOPs/byte} \quad (N_r \ll d_{\text{in}}, d_{\text{out}}). \tag{15}$$

The folded GEMM is compute-bound when $\mathcal{I}(N_r) \geq \mathcal{I}^*$, the accelerator's peak-FLOPs-to-HBM-bandwidth ratio (Tab. 2). In this regime an FLOPs reduction of $(1 - r_i)$ yields a near-proportional latency reduction. Below it, weight-load traffic dominates: SPECFOLD's analytical savings translate to latency only to the extent that the implementation amortises weight reads and avoids materialising the parent–child copies of Eqs. 9 and 12 through HBM.

GPU (bf16, dense)	TFLOPS	HBM TB/s	$\mathcal{I}^*$
A100 80GB	312	2.04	153
H100 SXM5	989	3.35	295
H200 SXM5	989	4.80	206

Table 2: Break-even arithmetic intensity (FLOPs/byte) for bf16 dense matmul on representative datacenter GPUs.

C.2 Empirical confirmation

Tab. 3 reports the per-call latency and achieved tensor-core throughput of the Fast-dLLM-v2-7B SwiGLU MLP at varying N_r on H200 bf16. Latency is approximately constant for $N_r \leq 128$ (bandwidth-bound: the $3{\times}130$ MB projection weights are streamed from HBM regardless of N_r) and grows linearly for $N_r \geq 256$ (compute-bound).

N_r	Latency (μs)	TFLOPS	% peak
16	128	51	5 %
40	130	125	13 %
64	127	205	21 %
128	140	372	38 %
288	212	553	56 %
512	309	674	68 %

Table 3: Dense Fast-dLLM-v2-7B SwiGLU MLP on H200 SXM5 (bf16): per-call latency and achieved compute throughput vs. recomputed-row count N_r. Averaged over 20 warmed-up CUDA-event timings.

Hardware utilisation rises from 5 % of peak at $N_r{=}16$ to 68 % at $N_r{=}512$, empirically confirming the regime boundary predicted by Eq. 15.

C.3 Two operating regimes for SPECFOLD

(i) Compute-bound configurations. Larger blocks or more drafts ($N{\geq}64$ or $B{\geq}16$) push the per-layer recomputed-row count into the linear region of Tab. 3 ($\mathcal{I}(N_r) \geq \mathcal{I}^*$), where the analytical FLOPs reductions in Tab. 1 translate directly to latency improvement with any standard cuBLAS/CUTLASS backend. **(ii) Bandwidth-bound configurations.** At the conservative settings used in our main experiments ($N{=}32$, $B{=}8$, fold rate $\sim 85\%$, giving $N_r \approx 40$), the per-layer GEMM lies in the flat region of Tab. 3 and weight-load traffic is paid regardless of N_r. Realising SPECFOLD's FLOPs reduction as latency improvement in this regime requires the fused on-chip kernel.