SPECFOLD: FOLDING MULTI-BRANCH REDUNDANCY FOR FASTER SPECULATIVE DECODING IN DIFFUSION LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion large language models (DLLMs) generate text through iterative block denoising, and multi-branch 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 identify a complementary redundancy axis within each speculative verification pass: multi-branch computational redundancy. During speculative verification, draft branches inherit most tokens from their parents while unmasking a small set of additional positions, causing large portions of hidden states to remain highly similar across branches. We propose SPECFOLD, an algorithm–system co-design that exploits this multi-branch redundancy to reduce the cost of multi-branch speculative verification. Algorithmically, SPECFOLD performs token-level residual gating and selectively reuses parent computation through folded attention and FFN while preserving residual hidden states. Systemically, a Triton kernel implementation translates this fine-grained reuse into end-to-end throughput gains through efficient sparse multi-branch execution. SPECFOLD is orthogonal to temporal caching and compatible with existing DLLM speculation strategies. Across two DLLM families, five models, and five standard benchmarks, SPECFOLD achieves up to $1.59\times$ throughput over Spiffy and up to $1.95\times$ over vanilla decoding, while maintaining comparable 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. (2025); Arriola et al. (2025). The paradigm has scaled from small-scale block-diffusion prototypes Arriola et al. (2025), through dense models Nie et al. (2025); Ye et al. (2025); Wu et al. (2025); Fu et al. (2026), to larger sparse Mixture-of-Experts (MoE) models Zhu et al. (2025); Bie et al. (2025; 2026). Despite this block-level parallelism, DLLM decoding remains costly because high-quality generation requires repeatedly evaluating the target model to refine the active denoising block. Even 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. (2025); Sun et al. (2026), these repeated target-model evaluations remain the dominant cost Gao et al. (2025); Wu et al. (2025).

Speculative decoding offers a complementary route to reducing the number of denoising calls. In AR 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). For latency-sensitive generation, *multi-branch* variants further organize candidate continuations into draft graphs and verify multiple branches in parallel, trading additional parallel computation for fewer sequential decoding steps. 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. (2026); Li et al. (2025); Gao et al. (2025). This adaptation is structurally different from AR speculation. First, DLLM attention is bidirectional, so each draft is a full block state rather than a next-token continuation. Second, draft branches across denoising steps form a directed graph of related masked-token configurations. Existing *multi-branch* speculative DLLM

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

methods verify one main branch and B draft branches in one pass. If a draft matches the subsequent main-branch denoising trajectory, its precomputed logits allow skipping one or more future denoising calls. Concretely, the decoding cost can be decomposed as

$$\underbrace{\text{Total decoding cost}}_{\text{End-to-end}} = \underbrace{\text{number of model calls}}_{\text{❶ Speculation}} \times \underbrace{\text{cost per model call}}_{\text{❷ Multi-branch execution}}. \tag{1}$$

While multi-branch speculative decoding reduces ❶ in Eq. 1, it also introduces a new bottleneck: each forward pass densely evaluates the main branch together with all draft branches, increasing cost per model call ❷ as the branch count grows. The key observation of this paper is that this dense forward pass contains substantial *multi-branch redundancy*. By construction, a draft inherits almost all tokens from its parent and differs only at a small set of speculative unmasked positions. Consequently, most token representations in the child draft remain nearly identical to the corresponding parent representations across Transformer layers. Our profiling on representative block-diffusion DLLMs demonstrates that more than 69% of measured draft–layer–position–step tuples have large parent–child hidden-state similarity, with divergence concentrated near recently unmasked positions. However, existing multi-branch speculative DLLM methods process every draft branch densely, recomputing layers for representations that are already available from the parent draft Agrawal et al. (2026); Li et al. (2025); Gao et al. (2025).

To exploit this redundancy, we propose SPECFOLD, an algorithm–system co-design for efficient multi-branch speculative DLLM decoding, reducing ❷ in Eq. 1, the cost of each cross-branch execution call in multi-branch speculative DLLM decoding. At the algorithm level, SPECFOLD builds a dynamic computation reuse relation across branches via a token-level parent–child residual gate. Similar positions reuse parent computation, which we refer to as *folding* in the following text, while divergent positions are recomputed. Because the bi-directional attention in DLLM couples all positions, SPECFOLD reuses the parent's streaming attention state and corrects it with child's recomputed keys and values. FFN outputs are reused at folded token positions, and each draft maintains its residual streams to preserve and accumulate cross-branch diversity. At the system level, a Triton kernel implementation compacts recomputed token positions across branches, resolves fold chains, performs projection and FFN computation with packed recomputed positions, and applies the attention correction without materializing redundant branch tensors. Together, the algorithm exposes fine-grained cross-branch reuse and the system converts it into lower per-model-call cost ❷ in Eq. 1.

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

- We identify *multi-branch activation redundancy* as a new efficiency opportunity in speculative DLLM decoding. Unlike temporal redundancy across denoising steps, this redundancy arises within a single speculative verification step.
- We introduce SPECFOLD, an algorithm–system co-design for folded multi-branch speculative DLLM decoding. At the algorithm level, SPECFOLD combines token-level residual gating, layer-wise hidden-state reuse, folded FFN, and attention correction, while maintaining branch hidden-state diversity. At the system level, a Triton kernel implementation maps these reuse decisions to sparse GPU execution that avoids redundant branch computation. The design is agnostic to the underlying draft-construction policy (Sec. 4.3), orthogonal to temporal caching as used in Fast-dLLM-v2, and compatible with existing speculative DLLM methods (Sec. 4).
- We show that SPECFOLD achieves up to $1.59\times$ throughput over Spiffy and $1.95\times$ over vanilla decoding across two DLLM families (Fast-dLLM-v2, Nemotron-Labs-Diffusion), five models, and five benchmarks, without compromising 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, as illustrated in Fig. 1-(a). Unlike autoregressive (AR) 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

$$F_\theta(C_k, X_k^t) \to p_\theta(\cdot \mid C_k, X_k^t), \tag{2}$$

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. (2025); 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. Multi-branch speculative DLLM decoding instead attempts to skip future denoising calls entirely.

2.2 MULTI-BRANCH SPECULATIVE DECODING IN DLLMS

Multi-branch 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). For latency-sensitive generation, *multi-branch* variants further organize candidate continuations into draft graphs and verify multiple branches in parallel, trading additional parallel computation for fewer sequential decoding steps. Adapting this idea to DLLMs requires a different structure because DLLM attention is bidirectional. In DLLM, a draft is a full active-block state with a particular set of unmasked and masked positions.

At a multi-branch speculative DLLM verification step (see Fig. 1-(b)), 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 rooted at the main branch. Each draft branch i has a parent draft $\mathrm{par}(i) \in \{0, \ldots, i-1\}$, so every branch's parent chain traces back to the root $X_{0,k}^t$; it inherits the parent draft state and additionally unmasks at most S_t positions. Then, the target DLLM verifies all branches in a single batched forward pass:

$$p_{i,k}^t = F_\theta(C_k, X_{i,k}^t), \qquad i = 0, 1, \ldots, B. \tag{3}$$

The resulting draft–logit pairs $\{(X_{i,k}^t, p_{i,k}^t)\}_{i=1}^B$ are stored in a 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 model 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. (2026); Li et al. (2025); Gao et al. (2025).

Existing multi-branch speculative DLLM methods differ in how they construct drafts and define acceptance, but commonly execute verification as a dense batch that computes every position of every branch Agrawal et al. (2026); Li et al. (2025); Gao et al. (2025); we refer to this execution strategy as *dense verification*. This reduces ❶ in Eq. 1, but increases ❷ by evaluating all branches densely within each model call.

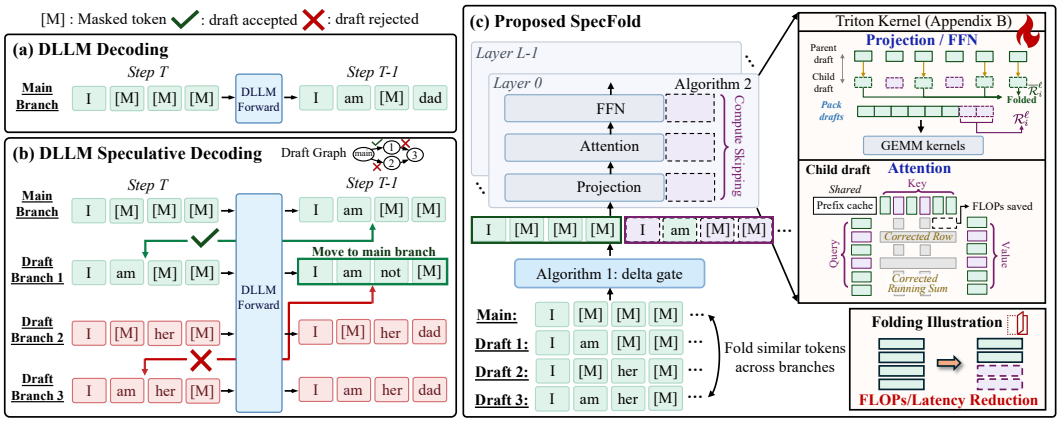

Figure 1: Overview of SPECFOLD. (a) Standard DLLM decoding iteratively unmasks tokens through repeated forward passes. (b) Multi-branch speculative DLLM decoding verifies a main branch together with several draft branches in one batched forward pass. If a draft matches a subsequent main-branch state, its precomputed outputs are reused to skip a future DLLM evaluation. (c) SPECFOLD exploits cross-branch redundancy within each verification pass. Before every Transformer layer, a parent–child residual gate identifies folded positions, which reuse parent computation, while divergent positions are recomputed through the folded attention and FFN execution, improving decoding throughput.

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 model 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). Multi-branch speculative decoding (Miao et al., 2024; Cai et al., 2024; Li et al., 2024) batches a draft graph of continuations into one verification pass and reuses the shared-prefix KV cache across graph nodes, but recomputes every sibling branch's suffix activations independently. SPECFOLD targets an unexploited third axis: *multi-branch redundancy* within the DLLM verification pass for speculative drafts, folding the computation of draft branches, without modifying the draft-construction policy or model weights.

3 THE PROPOSED SPECFOLD FRAMEWORK

SPECFOLD builds on the algorithmic observation that multi-branch speculative-verification DLLM decoding exhibits strong cross-branch redundancy that dense computation fails to exploit. 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. 3 is redundant. We characterize this redundancy (Sec. 3.1), present the SPECFOLD algorithm (Sec. 3.2), and describe the Triton kernel implementation that translates reused computation into throughput gains (Sec. 3.3).

For notations, we write $\mathbf{H}^{\ell}_{i,k} \in \mathbb{R}^{N\times d}$ for the hidden state at input of Transformer layer ℓ, denoising block k, and branch i, with $\mathbf{H}^{\ell}_{i,k}[n]$ denoting token position n. A per-layer gate on the parent-child hidden state residual magnitude determines folding for both the attention and FFN layers. Fig. 1-(c) summarizes SPECFOLD. Within a Transformer layer, token-level residual gating identifies folded token positions, parent branch computation is reused for projection and FFN, attention is corrected for child-specific updates, and the algorithmic sparsity is mapped to Triton kernels for higher decoding throughput.

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 $X_{\mathrm{par}(i),k}$. Therefore, we expect the corresponding layer-input

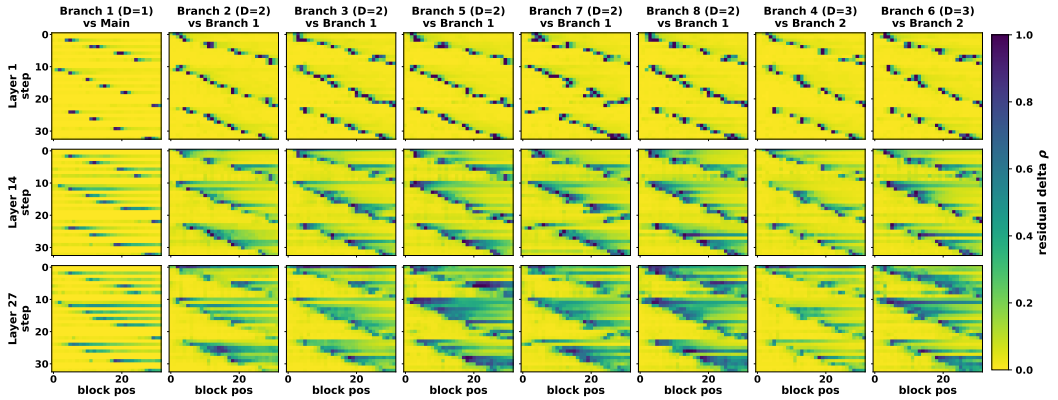

Figure 2: Per-token relative residual ρ of the layer-input hidden state between each draft branch and its immediate parent branch in the draft graph, recorded during the speculative verification passes of Fast-dLLM-v2-7B decoding a GSM8K sample. Columns are the 8 draft branches (D denotes a branch's depth in the draft graph), and rows are layers spanning the model depth. Within a panel, each row is one denoising step and each column one block position. The color indicates the level of divergence.

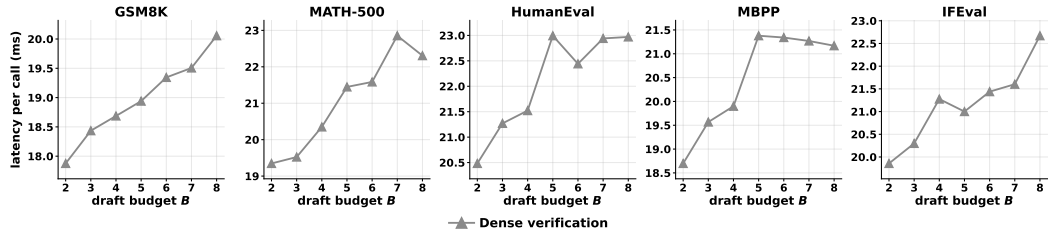

Figure 3: Mean latency per model call of dense verification versus draft budget B on Nemotron-Labs-Diffusion-3B on H100.

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 forward pass on branch $X_{i,k}$ repeats compute that was already done for the parent, $X_{\mathrm{par}(i),k}$.

Empirical profile. We test these predictions on redundancy during real speculative verification passes of Fast-dLLM-v2-7B (block size $N = 32$, calibrated draft graph with $B = 8$ drafts). Fig. 2 visualizes the relative residual $\rho^{\ell}_{i,k,n}$ between $\mathbf{H}^{\ell}_{i,k}[n]$ and $\mathbf{H}^{\ell}_{\mathrm{par}(i),k}[n]$ (defined in Eq. 4) across drafts (i), layers (ℓ), denoising steps, and token positions (n) within the block. We conclude the following observations:

Observation 1: Hidden states match across parent–child pairs. Across all 28 layers, 69% of (draft, step, layer, position) cells satisfy $\rho \leq 0.1$ and 86% satisfy $\rho \leq 0.3$. High-divergence regions concentrate along the unmasking positions and the positions draft i unmasked, 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.

Observation 2: Dense verification pushes diffusion iteration to a compute-heavy workload. We implemented Spiffy Agrawal et al. (2026) as a dense-verification baseline for DLLM decoding. Fig. 3 plots the mean latency per model call on Nemotron-Labs-Diffusion-3B as the draft budget grows from B=2 to B=8. Although the weights stream once for the whole batch, per-call latency rises with every added draft branch across the five benchmarks, demonstrating that dense verification computes every position of every draft, paying latency per model call.

3.2 SpecFold: Folding Redundant Drafts

SpecFold exploits the cross-branch redundancy characterized in Sec. 3.1. As stated in Observation 1, for a parent–child draft pair $(\mathrm{par}(i),\ i)$ among the speculative drafts, the hidden states $\mathbf{H}^{\ell}_{i,k}[n]$ at inherited token positions are nearly identical across all layers. Rather than recomputing these redundant activations, SpecFold identifies them via a per-token parent–child relative-residual gate

Algorithm 1 One SPECFOLD step.

Require: Branches $\{X_{i,k}^t\}_{i=0}^B$, recompute threshold δ, prefix KV cache $\mathcal{C}$, LM head W_{LM}, number of layers L
Ensure: Output logits $\{p_{i,k}^t\}_{i=0}^B$ for all branches i

Embed all branches: $\mathbf{H}_{i,k}^0 \leftarrow \mathrm{Embed}(X_{i,k}^t)$ for all $i \in \{0, \dots, B\}$
for each layer $\ell = 0, \dots, L-1$ **do**
  $\overline{\mathcal{R}}_0^\ell \leftarrow [N]$ ▷ main branch always recomputes
  for each draft $i = 1, \dots, B$ **do**
    $\overline{\mathcal{R}}_i^\ell \leftarrow \{\, n : \|\mathbf{H}_{i,k}^\ell[n] - \mathbf{H}_{\mathrm{par}(i),k}^\ell[n]\| > \delta \, \|\mathbf{H}_{\mathrm{par}(i),k}^\ell[n]\| \,\}$ ▷ gate on hidden-state residuals
    $\mathcal{R}_i^\ell \leftarrow [N] \setminus \overline{\mathcal{R}}_i^\ell$ ▷ folded / recompute sets
  end for
  $\{\mathbf{H}_{i,k}^{\ell+1}\}_{i=0}^B \leftarrow \textsc{SpecFoldLayer}(\{\mathbf{H}_{i,k}^\ell\}_{i=0}^B, \mathrm{par}(\cdot), \{\overline{\mathcal{R}}_i^\ell\}_{i=0}^B, \mathcal{C}_\ell)$ ▷ Alg. 2
end for
$p_{i,k}^t \leftarrow W_{\mathrm{LM}} \, \mathrm{Norm}(\mathbf{H}_{i,k}^L)$ for all $i \in \{0, \dots, B\}$
Cache $\{(X_{i,k}^t, p_{i,k}^t)\}_{i=0}^B$ for speculative verification

before the QKV projection at each Transformer layer, and folds the child's computation into the parent's at those positions.

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

$$\rho_{i,k,n}^\ell \;=\; \frac{\left\|\mathbf{H}_{i,k}^\ell[n] - \mathbf{H}_{\mathrm{par}(i),k}^\ell[n]\right\|}{\left\|\mathbf{H}_{\mathrm{par}(i),k}^\ell[n]\right\|}. \tag{4}$$

Given a recompute threshold δ, if $\rho_{i,k,n}^\ell \le \delta$, token n of draft i is *folded* at layer ℓ: its query, key, and value are inherited from the parent, and its attention and FFN outputs are obtained via the folded computation. The resulting folded ($\mathcal{R}_i^\ell$) and recomputed ($\overline{\mathcal{R}}_i^\ell$) position sets are:

$$\mathcal{R}_i^\ell \;=\; \left\{n : \rho_{i,k,n}^\ell \le \delta\right\}, \quad \overline{\mathcal{R}}_i^\ell \;=\; [N] \setminus \mathcal{R}_i^\ell. \tag{5}$$

Alg. 1 summarizes one complete SPECFOLD step. We highlight a few design choices that materialize the throughput gain from the exploitation of cross-branch redundancy, while maintaining the hidden state diversity across speculative branches:

- **Maintaining diversity through residual path.** Across Transformer layers, each position in each draft branch maintains its residual stream since the residual add costs little computation. This design allows the hidden state diversity across branches to be propagated across layers.
- **Maintaining diversity through cross-position attention.** At self-attention layers, the attention output at a folded query position remains exact with respect to the branch's own KV cache by correcting the parent's streaming attention state, efficiently implemented with details in Sec. 3.3 and Appendix A.1. Across layers, each position's residual stream keeps accumulating its own branch's cross-position attention contributions through the residual add. Once these contributions push the divergence measure, $\rho_{i,k,n}^\ell$ above δ, position n recomputes, preserving branch diversity.
- **Cutting redundancy at projections and FFN layers.** The QKV projection and FFN layers are position-wise, so folded positions inherit the parent's outputs. These layers only run with the $|\overline{\mathcal{R}}_i^\ell|$ recomputed positions.

In Alg. 1, there is a SPECFOLDLAYER procedure that materializes the computational reduction. See Appendix A for the detailed implementation.

3.3 TRITON IMPLEMENTATION

To materialize the algorithmic computation reduction into actual end-to-end throughput gains, we implement each SPECFOLDLAYER as a set of efficient Triton kernels. For gating in Eq. 4, a compact kernel evaluates the residual, records the source of folded positions, and compacts all recomputed

Table 1: SPECFOLD evaluation, reporting per-sample NFE, decoding throughput (tokens/s), and accuracy across five benchmarks and five model scales spanning the Fast-dLLM-v2 and Nemotron-Labs-Diffusion families. SPECFOLD uses recompute threshold $\delta = 0.1$. Nemotron-Labs-Diffusion models run in diffusion mode. The MATH row uses the MATH-500 test set.

Benchmark	Method	Fast-dLLM-v2-1.5B			Fast-dLLM-v2-7B			NLD-3B			NLD-8B			NLD-14B		
		NFE↓	TPS↑	Acc.↑	NFE↓	TPS↑	Acc.↑	NFE↓	TPS↑	Acc.↑	NFE↓	TPS↑	Acc.↑	NFE↓	TPS↑	Acc.↑
GSM8K	Vanilla	113	145	63.08%	113	170	82.41%	112	144	88.02%	128	102	84.84%	113	84	83.02%
	Spiffy	93	151	62.70%	98	163	83.32%	86	163	87.95%	93	118	84.84%	82	99	83.47%
	SPECFOLD	94	237 (↑63%)	61.64%	99	186 (↑10%)	81.80%	86	207 (↑44%)	88.40%	93	132 (↑29%)	84.31%	83	102 (↑21%)	82.64%
MATH	Vanilla	263	173	38.60%	229	160	57.80%	225	156	72.60%	297	113	83.80%	246	104	84.00%
	Spiffy	219	183	37.40%	195	160	57.60%	172	158	72.60%	219	118	82.20%	183	108	84.40%
	SPECFOLD	210	265 (↑53%)	36.80%	196	182 (↑14%)	56.00%	172	213 (↑37%)	73.20%	224	140 (↑23%)	83.20%	184	116 (↑11%)	85.00%
HumanEval	Vanilla	161	133	39.02%	156	117	65.85%	286	110	73.78%	464	86	66.46%	343	77	71.34%
	Spiffy	121	157	39.63%	117	122	64.63%	212	114	75.61%	313	95	67.68%	237	82	71.34%
	SPECFOLD	123	242 (↑83%)	40.85%	111	149 (↑27%)	61.59%	212	156 (↑42%)	72.56%	305	111 (↑29%)	67.07%	232	93 (↑21%)	73.78%
MBPP	Vanilla	153	121	48.68%	163	121	68.78%	255	113	73.02%	247	87	83.07%	299	81	79.89%
	Spiffy	118	148	48.41%	114	123	67.99%	191	124	70.37%	165	99	83.86%	206	92	80.42%
	SPECFOLD	120	236 (↑95%)	48.41%	120	156 (↑29%)	68.78%	189	162 (↑43%)	73.28%	169	113 (↑31%)	83.07%	205	97 (↑20%)	78.31%
IFEval	Vanilla	294	132	48.43%	270	111	64.88%	234	94	60.26%	313	96	59.89%	304	73	62.11%
	Spiffy	228	145	48.43%	202	110	65.06%	159	105	60.63%	215	105	58.60%	211	86	60.81%
	SPECFOLD	225	228 (↑73%)	46.21%	203	140 (↑27%)	63.96%	166	142 (↑52%)	59.15%	213	121 (↑26%)	59.89%	205	91 (↑24%)	61.37%

positions across branches. This packing allows QKV projections and FFNs to operate on this packed batch. A source-resolution kernel walks the fold chain, enabling every folded position to access the parent state.

For attention layers, we implement a FlashAttention-style streaming kernel Dao et al. (2022). For recomputed positions, query vectors execute streaming attention over the shared prefix and each branch's KV cache. For folded positions, query vectors reuse the streaming attention state of the parent and correct it for recomputed keys and values within the branch. The shared prefix KV cache is expanded across branches as a zero-copy strided view.

This implementation supports both evaluated model families, including grouped-query attention. Appendix B details the algorithm and kernel implementation.

4 EXPERIMENTS

4.1 SETUP

Models and baselines. We evaluate two DLLM families: Fast-dLLM-v2 Wu et al. (2025) (1.5B, 7B) and Nemotron-Labs-Diffusion (NLD) Fu et al. (2026) (3B, 8B, 14B). For Fast-dLLM-v2, we follow the default configuration with block size 32, small-block size 8, confidence threshold 0.9, and block caching disabled. For Nemotron-Labs-Diffusion, we use block size 32, confidence threshold 0.9, and refresh the prefix KV cache with a causal forward pass at each block boundary. Across all models, Spiffy constructs the draft graph and SPECFOLD reuses the same graph for fair comparison. We compare the following decoding strategies:

- *Vanilla decoding.* The native block-wise decoding of each model, without speculative drafts.
- *Spiffy* Agrawal et al. (2026). A calibration-based DLLM speculative decoding baseline. For each model–dataset pair, we use 50 calibration samples to construct a degree-1 draft graph, and instantiate B=7 draft branches at inference time according to the calibrated token-position and vocabulary-rank patterns.
- SPECFOLD. SPECFOLD is our proposed method with folded cross-branch computation, where we use the same draft graph construction algorithm as Spiffy.

Benchmarks, Metrics, and Hardware. 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, and HumanEval and MBPP are evaluated with EvalPlus Liu et al. (2023). We report three primary metrics: Number of Function Evaluations (NFE) per sample, tokens per second (TPS), and accuracy. NFE measures the number of denoising steps required for generation, while TPS measures end-to-end decoding speed on hardware. All experiments, unless otherwise stated, are run on a single NVIDIA H100 80GB PCIe GPU.

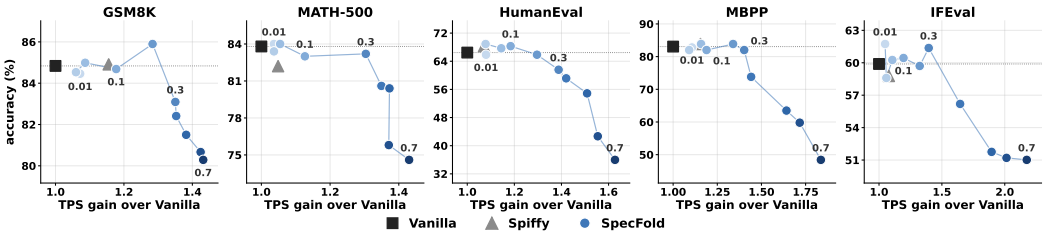

Figure 4: Recompute-threshold sweep of SPECFOLD on Nemotron-Labs-Diffusion-8B (H100, B=8). Each panel plots task accuracy against the throughput gain (TPS) over Vanilla as δ increases from 0.01 to 0.7 (darker markers = larger δ, i.e., more folding); the dotted line marks the Vanilla accuracy. Upper-right is better.

4.2 MAIN RESULTS

Table 1 presents our primary results across all five models. Both efficiency metrics (NFE, TPS) are reported relative to Vanilla decoding: NFE reductions capture the denoising calls skipped by speculative verification, while TPS gains capture the end-to-end effect of SPECFOLD folding the compute of dense verification.

Fast-dLLM-v2. SPECFOLD reduces the cost of dense verification at both model scales. Across the five benchmarks, it achieves up to $1.59\times$ throughput over Spiffy and $1.95\times$ over Vanilla on Fast-dLLM-v2-1.5B, and up to $1.27\times$ over Spiffy and $1.29\times$ over Vanilla on Fast-dLLM-v2-7B. These results show that SPECFOLD converts the NFE savings of multi-branch speculative decoding into end-to-end speed by removing redundant computation introduced by dense verification while maintaining comparable accuracy. The accuracy drifts in Spiffy are floating-point noise, where we detailed this effect in Appendix C and justified the correctness of our implementation with a serialized-execution sanity check.

Nemotron-Labs-Diffusion. Across the 3B, 8B, and 14B models, SPECFOLD achieves up to $1.37\times$, $1.19\times$, and $1.13\times$ throughput over Spiffy, respectively, and up to $1.52\times$, $1.31\times$, and $1.24\times$ over Vanilla. The accuracy of SPECFOLD remains comparable to both Vanilla and Spiffy across all five benchmarks, with small deviations in both directions. Overall, the results show that SPECFOLD provides consistent end-to-end throughput gains across block-diffusion model families and scales without compromising accuracy.

4.3 ABLATIONS

Recompute-threshold sweep. We evaluate the sensitivity of SPECFOLD to the recompute threshold δ on Nemotron-Labs-Diffusion-8B across all five benchmarks, sweeping $\delta \in \{0.01, 0.02, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7\}$ while keeping all other decoding hyperparameters fixed. Fig. 4 plots the resulting accuracy–throughput trade-off per benchmark, with Vanilla and Spiffy shown as reference points.

SPECFOLD exhibits a broad and robust operating region in the latency-accuracy trade-off relation. Across all five benchmarks, accuracy stays at Vanilla's level through $\delta = 0.2$–0.3 while folding already delivers a 1.3–$1.4\times$ throughput gain (TPS) over Vanilla. Accuracy declines beyond this plateau. Measured throughput continues to grow with δ, but gains beyond $\delta = 0.3$ come at the cost of accuracy. Moreover, when δ approaches 0, forcing all drafts to be recomputed, the accuracy–throughput point approaches Spiffy across all benchmarks, validating our implementation.

These results indicate that the efficiency gains do not depend on a precise choice of δ. For the main evaluation in the paper, we fix $\delta = 0.1$ as a conservative default across all models and benchmarks without per-task tuning.

Draft-budget sweep. We analyze the sensitivity and effectiveness of SPECFOLD to the number of speculative draft branches B on Nemotron-Labs-Diffusion-3B across all five benchmarks, sweeping $B \in \{2, \ldots, 8\}$. Each budget uses its own calibrated draft graph with B nodes. Fig. 5 plots the throughput gain (TPS) over Vanilla of Spiffy and SPECFOLD at every B.

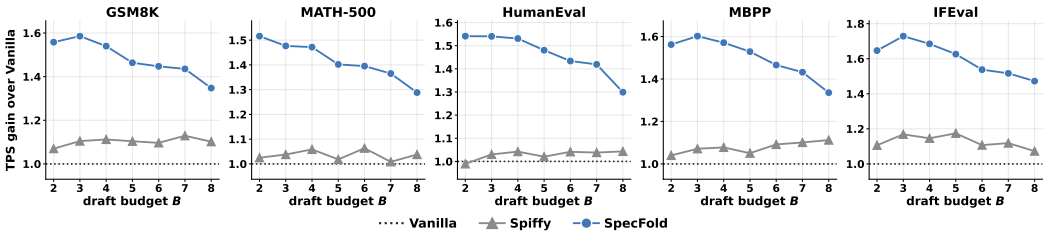

Figure 5: Draft-budget sweep on Nemotron-Labs-Diffusion-3B, fixing δ=0.1. Each panel plots the throughput gain (TPS) over Vanilla against the draft budget B. The gap between the two curves demonstrates the effect of folding.

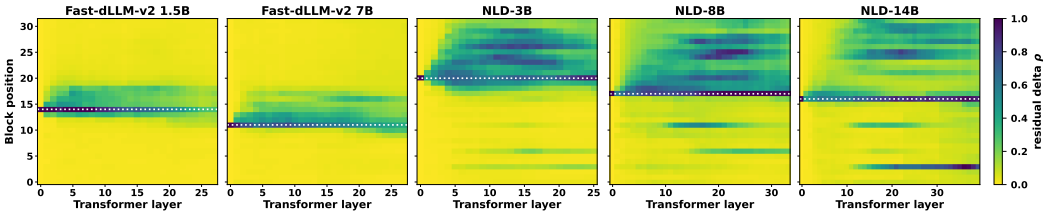

Figure 6: **Per-position relative residual between a parent draft and a child draft that unmasks a single token.** Each model decodes a GSM8K prompt to the middle of a block: the parent draft is the block state with 17 of the 32 positions committed, and the child draft adds one more unmasked token being the target's next most-confident masked position (the "edited position", white dotted line). Heatmaps show the per-position parent–child relative residual ρ_n^ℓ (Eq. 4) across layers.

Generally, the acceptance of speculative drafts grows with B and saturates around $B = 6$, where NFE reductions reach 30% relative to Vanilla. Throughput, however, distinguishes the effectiveness of SPECFOLD. Spiffy stays within 0.99–$1.17\times$ of Vanilla's TPS across the entire sweep due to the cost of dense verification. SPECFOLD, which shares the identical drafts, hit rates, and NFE at every B, converts the same acceptance into up to $1.73\times$ throughput gain. Appendix D.1 repeats this sweep on the remaining four models and demonstrates the same structure across scales and families. Furthermore, Appendix D.2 varies the draft-graph depth at a fixed budget and shows that SPECFOLD's advantage is consistent across drafting strategies.

4.4 ANALYSIS OF THE RESIDUAL RIPPLE

To test whether the layer-input hidden state $\mathbf{H}_{i,k}^{\ell}[n]$ provides a reliable basis for the folding decision in Eq. 4, we compare a parent-child $(\mathrm{par}(i), i)$ pair that differs by only one unmasked token in the speculative draft branch. Fig. 6 shows the structure of resulting parent–child relative residual ρ_n^ℓ.

The edited position creates a persistent high-residual difference across layers. Interestingly, the residual difference elsewhere splits by decoding state. Residual divergence ρ stays low at committed positions, while the divergence ripple concentrates in the masked positions and recently unmasked positions. This observation motivates a per-token gate in Eq. 4, providing folding decisions in a higher granularity.

5 CONCLUSION

We propose SPECFOLD, an algorithm–system co-design for accelerating multi-branch speculative DLLM verification by exploiting *cross-branch activation redundancy*. SPECFOLD combines token-level folding with a dedicated kernel implementation to reduce redundant computation within each speculative-verification pass. Across two DLLM families, five models, and five benchmarks, SPECFOLD achieves up to $1.59\times$ throughput over Spiffy and up to $1.95\times$ over Vanilla decoding, while maintaining comparable task performance.

AI USE STATEMENT

In this work, we used generative AI tools for polish writing. We have not used generative AI tools for idea initiation and retrieval or discovery (e.g., finding related work), and usage of generative AI for dataset generation and formal mathematical proof is not applicable to this work. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This paper presents system, empirical and methodological contributions and does not involve human subjects, data collection, or the release of new datasets or models. We do not foresee any ethical concerns arising from this work.

REPRODUCIBILITY STATEMENT

In this paper, the implementation methodologies, algorithms of key components as well as experimental details, including the model, datasets, and hyperparameters, are included in the manuscript and appendix such that to the best of the authors' knowledge, is sufficient for reproduction. All the models and datasets we used are publicly available on platforms such as Hugging Face. We will release the code upon acceptance.

REFERENCES

Sudhanshu Agrawal, Risheek Garrepalli, Raghavv Goel, Christopher Lott, Fatih Porikli, and Mingu Lee. Structuring the future: Diffusion LLM speculative decoding via calibrated draft graphs. In *ICML 2026 Workshop on Structured Probabilistic Inference & Generative Modeling*, 2026. URL https://openreview.net/forum?id=LHwa2TCan6.

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

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

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.

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

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

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, 2024.

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

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, 2021.

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

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

Yonggan Fu, Lexington Whalen, Abhinav Garg, Chengyue Wu, Maksim Khadkevich, Nicolai Oswald, Enze Xie, Daniel Egert, Sharath Turuvekere Sreenivas, Shizhe Diao, Chenhan Yu, Ye Yu, Weijia Chen, Sajad Norouzi, Jingyu Liu, Shiyi Lan, Ligeng Zhu, Jin Wang, Jindong Jiang, Morteza Mardani, Mehran Maghoumi, Song Han, Ante Jukic, Nima Tajbakhsh, Jan Kautz, and Pavlo Molchanov. Nemotron-labs-diffusion: A tri-mode language model unifying autoregressive, diffusion, and self-speculation decoding. Technical report, NVIDIA, 2026. Technical report.

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

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

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

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

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 ACM SIGOPS 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

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

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

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

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

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

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

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

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. 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*, pp. 932–949, 2024.

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.

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

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

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, pp. 33038–33046, 2026.

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

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

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. *arXiv preprint arXiv:2509.26328*, 2025.

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

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.

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. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

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

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

APPENDIX

- Sec. A: Folded Transformer Algorithm
  - Folded Self-attention
  - Folded FFN
  - Putting it Together: One Folded Layer
- Sec. B: Triton Implementation
- Sec. C: On the Numerical Discrepancy with Vanilla Decoding
- Sec. D: Additional Ablations
  - Draft-budget Sweep across Model Scales and Families
  - Ablation on Drafting Strategy
  - Ablation on Hardware

A FOLDED TRANSFORMER ALGORITHM

This appendix develops the SPECFOLDLAYER in detail. Following the notations in the main text, given the folded position set $\mathcal{R}_i^\ell$ and recomputed set $\overline{\mathcal{R}}_i^\ell$ from Eq. 5, we describe how each Transformer sublayer is restructured to avoid redundant computation. For brevity we drop the block index k and layer index ℓ, 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 $\overline{\mathcal{R}}_i^\ell$ as we do a single layer's derivation.

Per-branch state. Within a layer, each branch i owns three per-position tables over the N block positions: (1) the RoPE-transformed projections Q_i, K_i, V_i, (2) the streaming-softmax attention state (m_i, Z_i, O_i) (per-row running maximum, denominator, and normalized output, as maintained by FlashAttention), and (3) the FFN output table F_i. A branch initializes all three tables by *inheriting* its parent's and overwrites only the positions in $\overline{\mathcal{R}}_i$ with recomputed values. The main branch $i{=}0$ has no parent and recomputes every position ($\overline{\mathcal{R}}_0 = [N]$). Crucially, the hidden states $\mathbf{H}_i$ themselves are never overwritten with parent copies, accumulating cross-branch divergence.

The approximation enters in two places, both position-wise and gated by $\rho_{i,k,n}^\ell \le \delta$. First, a folded position inherits the parent's (q, k, v) instead of projecting its own hidden state, approximating the self-attention while allowing computation reuse from parent. Second, a folded position reuses the parent's FFN output (Sec. A.2).

A.1 FOLDED SELF-ATTENTION

The self-attention in DLLM is bidirectional and not position-wise. The output at position n depends on every key and value in the sequence. Thus, a folded position has to correct its attention values regarding the recomputed positions. Therefore, SPECFOLD maintains, per head and per branch, the streaming-softmax state over the concatenated prefix and block KV caches, and updates the folded positions.

Setup. Let $(K_\mathrm{P}, V_\mathrm{P})$ be the prefix KV cache of length P, shared by all branches, and let q_n^i, k_n^i, v_n^i denote branch i's per-head table rows. Write $s_{nj}^i = q_n^i \cdot k_j^i / \sqrt{d_h}$ for the score of query row n against key j of the concatenated table $[K_\mathrm{P} \,\|\, K_i] \in \mathbb{R}^{(P+N)\times d_h}$. The attention state of row n is

$$m_i[n] = \max_j s_{nj}^i, \qquad Z_i[n] = \sum_j e^{\,s_{nj}^i - m_i[n]}, \qquad O_i[n] = \frac{1}{Z_i[n]} \sum_j e^{\,s_{nj}^i - m_i[n]}\, v_j^i. \tag{6}$$

Recomputed query positions. For $n \in \overline{\mathcal{R}}_i$, the branch computes projections from its own hidden state, $(q_n^i, k_n^i, v_n^i) = \mathrm{RoPE}\big(W_{QKV}\,\mathrm{RMSNorm}(\mathbf{H}_i[n])\big)$, scatters them into its tables, and evaluates Eq. 6 with a FlashAttention pass over $[K_\mathrm{P} \,\|\, K_i]$ and $[V_\mathrm{P} \,\|\, V_i]$.

Folded query positions. For $n \in \mathcal{R}_i$, the position inherits the parent's projections,

$$(q_n^i,\, k_n^i,\, v_n^i) \leftarrow (q_n^p,\, k_n^p,\, v_n^p), \tag{7}$$

which is the sole attention-side approximation. The parent's state $(m_p[n], Z_p[n], O_p[n])$ is the exact softmax attention of the (now shared) query q_n^p over the *parent's* table. Since the child's table differs from the parent's only on the positions in $\overline{\mathcal{R}}_i$, the parent's state can be converted into the child's exactly, by removing the parent's contributions at those positions and adding the child's. With the rescaled maximum $m' = \max\big(m_p[n],\, \max_{e\in\overline{\mathcal{R}}_i} q_n^p \cdot k_e^i/\sqrt{d_h}\big)$ and $f = e^{\,m_p[n]-m'}$,

$$\begin{aligned}
a_\mathrm{old}(e) &= e^{\,q_n^p\cdot k_e^p/\sqrt{d_h} - m'}, \qquad a_\mathrm{new}(e) = e^{\,q_n^p\cdot k_e^i/\sqrt{d_h} - m'}, \qquad e \in \overline{\mathcal{R}}_i, \\
Z_i[n] &= Z_p[n]\, f \;-\; \textstyle\sum_e a_\mathrm{old}(e) \;+\; \sum_e a_\mathrm{new}(e), \\
O_i[n] &= \Big(O_p[n]\, Z_p[n]\, f \;-\; \textstyle\sum_e a_\mathrm{old}(e)\, v_e^p \;+\; \sum_e a_\mathrm{new}(e)\, v_e^i\Big) \,/\, Z_i[n].
\end{aligned} \tag{8}$$

Algorithm 2 SPECFOLDLAYER: one transformer layer with inherited QKV/FFN and exact incremental attention.

Require: Hidden states $\{\mathbf{H}_i\}_{i=0}^{B}$, parent map $\mathrm{par}(\cdot)$, recompute sets $\{\overline{\mathcal{R}}_i\}_{i=0}^{B}$, prefix KV cache $(K_{\mathrm{P}}, V_{\mathrm{P}})$ of length P, layer weights $(W_{QKV}, W_O, W_{\mathrm{FFN}})$, head dim d_h

Ensure: Next-layer hidden states $\{\mathbf{H}'_i\}_{i=0}^{B}$

for each branch $i = 0, \ldots, B$ in topological order **do** ▷ parents before children; kernel runs all branches in one order-free launch
  $(Q_i, K_i, V_i) \leftarrow (Q, K, V)_{\mathrm{par}(i)}$; $(m_i, Z_i, O_i) \leftarrow (m, Z, O)_{\mathrm{par}(i)}$; $F_i \leftarrow F_{\mathrm{par}(i)}$ ▷ inherit parent tables
  for $n \in \overline{\mathcal{R}}_i$ **do** ▷ recomputed positions: full QKV projection and attention
    $(Q_i[n], K_i[n], V_i[n]) \leftarrow \mathrm{RoPE}\big(W_{QKV}\,\mathrm{RMSNorm}(\mathbf{H}_i[n])\big)$
    $(m_i[n], Z_i[n], O_i[n]) \leftarrow \mathrm{FlashAttn}\big(Q_i[n],\, [K_{\mathrm{P}} \,\|\, K_i],\, [V_{\mathrm{P}} \,\|\, V_i]\big)$
  end for
  for $n \notin \overline{\mathcal{R}}_i$ **do** ▷ folded positions: correction of the inherited state
    $m' \leftarrow \max\big(m_i[n],\, \max_{e \in \overline{\mathcal{R}}_i} Q_i[n]\cdot K_i[e]/\sqrt{d_h}\big)$; $f \leftarrow e^{\,m_i[n]-m'}$
    $a_{\mathrm{old}}(e) \leftarrow e^{\,Q_i[n]\cdot K_{\mathrm{par}(i)}[e]/\sqrt{d_h}-m'}$, $a_{\mathrm{new}}(e) \leftarrow e^{\,Q_i[n]\cdot K_i[e]/\sqrt{d_h}-m'}$ **for** $e \in \overline{\mathcal{R}}_i$
    $Z' \leftarrow Z_i[n]\, f - \sum_e a_{\mathrm{old}}(e) + \sum_e a_{\mathrm{new}}(e)$
    $O' \leftarrow \big(O_i[n]\, Z_i[n]\, f - \sum_e a_{\mathrm{old}}(e)\, V_{\mathrm{par}(i)}[e] + \sum_e a_{\mathrm{new}}(e)\, V_i[e]\big) \,/\, Z'$
    $(m_i[n], Z_i[n], O_i[n]) \leftarrow (m', Z', O')$
  end for
  $\mathbf{H}_i^{+} \leftarrow \mathbf{H}_i + O_i\, W_O$ ▷ residual never folded
  for $n \in \overline{\mathcal{R}}_i$ **do**
    $F_i[n] \leftarrow \mathrm{FFN}\big(\mathrm{RMSNorm}(\mathbf{H}_i^{+}[n])\big)$ ▷ folded positions reuse $F_{\mathrm{par}(i)}$
  end for
  $\mathbf{H}'_i \leftarrow \mathbf{H}_i^{+} + F_i$
end for
return $\{\mathbf{H}'_i\}_{i=0}^{B}$

A.2 FOLDED FFN

After the attention state is assembled, the post-attention residual add is formed for *all* positions of every branch from the branch's own hidden state:

$$\mathbf{H}_i^{+}[n] = \mathbf{H}_i[n] + W_O\, O_i[n] \qquad \forall\, n \in \{0, \ldots, N{-}1\}. \tag{9}$$

As emphasized in Sec. 3.2, the residual stream allows every position, whether recomputed or folded, to accumulate and preserve position-wise divergence.

The FFN sublayer is position-wise, so folded positions can reuse the parent's output table directly:

$$F_i[n] = \begin{cases} F_p[n], & n \in \mathcal{R}_i, \\ \mathrm{FFN}\big(\mathrm{RMSNorm}(\mathbf{H}_i^{+}[n])\big), & n \in \overline{\mathcal{R}}_i, \end{cases} \qquad \mathbf{H}'_i[n] = \mathbf{H}_i^{+}[n] + F_i[n]. \tag{10}$$

Similar to the residual add at self-attention layer, $\mathbf{H}'_i[n]$ in Eq. 10 retains the branch's own hidden state, allowing the position to accumulate and preserve divergence.

A.3 PUTTING IT TOGETHER: ONE FOLDED LAYER

Alg. 2 composes the folded attention and folded FFN into a single SPECFOLDLAYER. The branches are visited in topological order such that every branch inherits fully updated parent tables and applies the one-hop correction of Eq. 8 against its immediate parent. The corrections propagate along fold chains, where the kernel implementation exploits this. It resolves each folded row to its nearest recomputed ancestor and applies the accumulated correction in one order-free launch over all branches (Appendix B).

B TRITON IMPLEMENTATION

This section maps the SPECFOLDLAYER of Appendix A.3 onto the Triton implementation. The implementation runs six kernel families per layer, in order, over a flattened index space of $(B{+}1)N$ (branch, position) pairs to realize the SPECFOLDLAYER algorithm described in Appendix A.

Table 2: Kernel pipeline for one SPECFOLDLAYER. $\overline{\mathcal{R}}$ denotes the union of the per-branch recompute sets $\overline{\mathcal{R}}_i^\ell$. The grid sizes are per-layer launch dimensions.

Kernel	Grid	Role
`delta_gate_compact`	$(B{+}1)N$	gate, source map, compaction, RMSNorm stats
`src_resolve`	$(B{+}1)N$	pointer-jump fold chains
`ln_qkv_rope_scatter`	$\lvert\overline{\mathcal{R}}\rvert$	norm + packed QKV + RoPE for recomputed positions
`flash_attn` (+state)	$\lvert\overline{\mathcal{R}}\rvert \times H$	recomputed-position attention; emits (m, Z, O)
`chain_correction`	$(B{+}1)N \times H$	correcting attention for folded positions
`ln_gateup_silu` / `down_residual_scatter`	$\lvert\overline{\mathcal{R}}\rvert$	fused FFN + residual add for recomputed positions

Gate and compaction. One program per (branch, position) pair evaluates the relative residual of Eq. 4 against the parent's corresponding position. Each program writes three outputs: a recompute flag, a source index (self if recomputed, the parent otherwise), and, via atomic counters, its flat index into two compacted lists: a global recomputed position list that the GEMM and attention kernels iterate over, and a per-branch list that the correction kernel uses to visit one branch's recomputed entries. The main branch bypasses the gate and is marked recompute at every position.

Source resolution. The sources form chains through the draft graph as a folded position can point at a position that the corresponding position at parent is also folded. A pointer-jumping kernel squares the source map $\lceil \log_2 \mathrm{depth} \rceil$ times, after which every folded position addresses the ancestor row that is not folded.

Projections. Recomputed positions are processed as one contiguous batch through a kernel that applies RMSNorm, multiplies by the QKV projection weights packed into a single matrix, applies rotary embeddings, and scatters the results into the full Q, K, V tables at their (branch, position) slots. Folded slots stay untouched.

Attention with state reuse. A FlashAttention-style kernel processes only recomputed query rows, streaming over the shared prefix cache and over the in-block KV tables. Besides the attention output, it retains each row's running softmax maximum m and denominator Z in per-head state tables. A second kernel serves folded positions: starting from the source row's (m, Z, O), it walks the fold chain hop by hop and applies the exact correction of Alg. 2, swapping the ancestor's key and value contributions for the current branch's recomputed ones at that branch's segmented entries only.

FFN and residual. A kernel applies RMSNorm, the gate and up projections packed into one GEMM, the SiLU product, the down projection, and the residual addition for recomputed rows, scattering results into the output table; folded positions gather the source row's FFN output and add their *own* residual stream, which is never folded. The layer's output table then feeds the next layer's gate directly.

The implementation is purely in Triton over HuggingFace checkpoints for both evaluated model families.

C On the Numerical Discrepancy with Vanilla Decoding

In principle, speculative decoding is equivalent in exact arithmetic to DLLM autoregressive decoding. Every accepted draft token is, by construction, the same token vanilla decoding would have produced from the same context (Sec. 2.2). With this theory, the Spiffy Agrawal et al. (2026) row in our metric tables should match the Vanilla baseline. 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.

The source of this discrepancy is introduced by numerical drifts in tensor operator libraries. As also documented by Spiffy Agrawal et al. (2026), `torch.bfloat16` matrix multiplication on CUDA GPUs can use an accumulation order that depends on the shape of the input tensor. Once speculative

decoding batches the main branch together with B draft branches, the matmul reduces 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 inevitable small accuracy drift we observe.

To justify that the discrepancy is caused by batched matmul numerics rather than an algorithmic implementation error, 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, so the accumulation order is preserved. In this serialized run, our implementation reproduces the Vanilla baseline metrics on every benchmark we tested. This rules out an algorithmic implementation error in the speculative verification logic.

D Additional Ablations

This section presents the additional ablations conducted for the SpecFold algorithm and implementation.

D.1 Draft-budget sweep across model scales and families

We extend the draft-budget sweep of Sec. 4.3 from Nemotron-Labs-Diffusion-3B to the four remaining models of Tab. 1 under the identical setup. Fig. 7 collects all four models in one grid.

Across models and benchmarks, SpecFold sustains a consistent throughput advantage over Spiffy. As the draft budget B grows, however, draft acceptance saturates while every added branch still adds verification cost. Therefore, the end-to-end throughput gain diminishes toward the dense baseline.

D.2 Ablation on Drafting Strategy

We extend our investigation in SpecFold's effectiveness on the choice of shape of draft graph and drafting strategy. We constrain the drafting strategy by capping the draft-graph depth at $D \in \{1, 2, 3, 4\}$ on Nemotron-Labs-Diffusion-3B, following settings in Table 1. This setup spans the two extremes of drafting strategy: $D{=}1$ yields seven sibling drafts that each differ from the main branch by a single token, and $D{=}4$ yields deep chains in which each draft extends its parent. Fig. 8 plots the throughput gain (TPS) over Vanilla for both methods at every depth.

We demonstrate that SpecFold's advantage is insensitive to the drafting strategy. It delivers a 1.31–$1.57\times$ throughput gain over Vanilla at every depth on every benchmark, with fold ratios of 0.52–0.67 across all shapes. This demonstrates that, while drafting impacts the acceptance of speculative drafts, SpecFold provides consistent throughput gains.

D.3 Ablation on Hardware

To test whether SpecFold's gains depend on a particular accelerator, we repeat the Nemotron-Labs-Diffusion-3B configuration of the sweeps above (recompute threshold $\delta = 0.1$, $B = 8$ draft branches, identical calibration graphs) on NVIDIA A100 GPUs and compare against H100 runs of the same configuration.

As shown in Tab. 3, SpecFold's relative gain is larger on A100 (1.57–$1.96\times$) than on H100 (1.29–$1.47\times$). Task accuracy is preserved. On the A100, which has lower compute capacity, the dense draft batch accounts for a larger share of each model call, so SpecFold recovers more throughput. In contrast, the Spiffy baseline, which pays the dense model call cost for every draft, stays within 0.93–$1.24\times$ TPS gain on both GPUs.

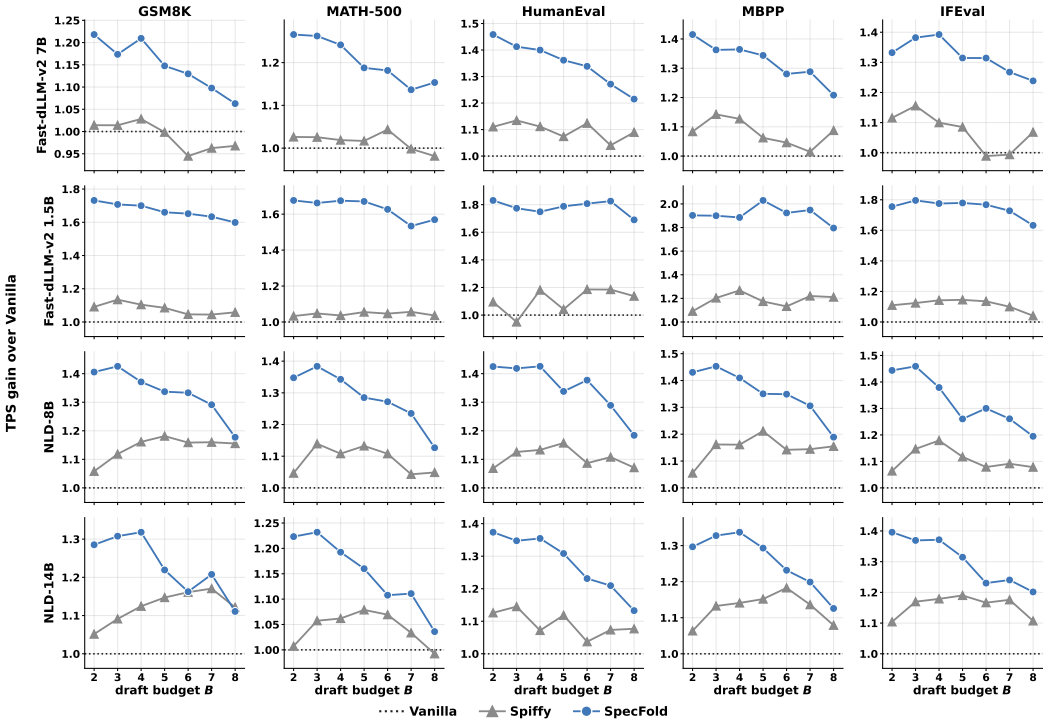

Figure 7: Draft-budget sweep on the four remaining models (rows) across the five benchmarks (columns), presented as in Fig. 5: each panel plots the throughput gain (TPS) of Spiffy and SPECFOLD over Vanilla against the draft budget B, with the dotted line marking Vanilla. At every B the two methods share identical drafts and NFE, so the gap between the curves is the effect of folding.

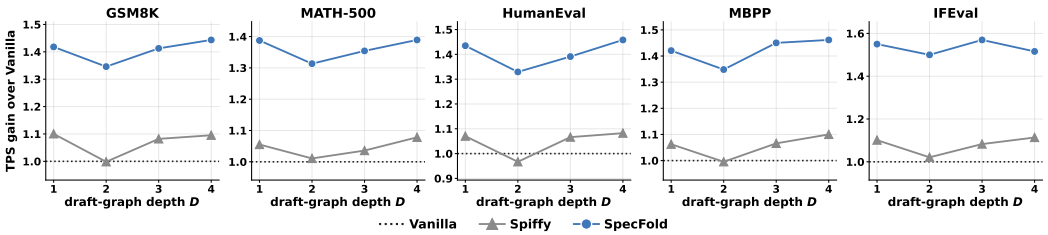

Figure 8: Drafting-strategy ablation on Nemotron-Labs-Diffusion-3B: throughput gain (TPS) over Vanilla versus the draft-graph depth cap D at fixed budget B=7, presented as in Fig. 5. Spiffy and SPECFOLD share identical drafts and NFE at every depth.

Table 3: GPU-type ablation on Nemotron-Labs-Diffusion-3B: decoding throughput (tokens/s), TPS gain over the same GPU's vanilla decoding, and task accuracy.

Benchmark	Method	A100			H100		
		TPS↑	TPS gain	Acc.↑	TPS↑	TPS gain	Acc.↑
GSM8K	Vanilla	65	1.00×	88.32%	144	1.00×	88.02%
	Spiffy	73	1.12×	88.02%	159	1.10×	87.95%
	SPECFOLD	110	**1.69**×	88.48%	194	1.35×	88.48%
MATH	Vanilla	73	1.00×	72.80%	156	1.00×	72.60%
	Spiffy	86	1.18×	74.20%	162	1.04×	72.60%
	SPECFOLD	132	**1.81**×	75.20%	201	1.29×	71.80%
HumanEval	Vanilla	51	1.00×	73.78%	110	1.00×	73.78%
	Spiffy	63	1.24×	74.39%	115	1.04×	75.61%
	SPECFOLD	98	**1.95**×	73.78%	143	1.30×	71.34%
MBPP	Vanilla	64	1.00×	70.37%	113	1.00×	73.02%
	Spiffy	60	0.93×	69.84%	126	1.11×	70.37%
	SPECFOLD	101	**1.57**×	70.90%	151	1.34×	71.69%
IFEval	Vanilla	46	1.00×	59.52%	94	1.00×	60.26%
	Spiffy	53	1.16×	60.07%	101	1.07×	60.63%
	SPECFOLD	90	**1.96**×	60.63%	138	1.47×	58.96%