TREES FROM MARGINALS: AUTOREGRESSIVE DRAFTING WITH FACTORIZED PRIORS

Anonymous authors
Paper under double-blind review

ABSTRACT

On-device applications of autoregressive large language models (LLMs) require fast token generation for individual requests, yet sequential decoding limits generation speed while underutilizing available compute. Speculative decoding trades additional computation for more tokens per target-model forward pass. While factorized drafters have set new records in speculative decoding efficiency by predicting future-token marginals in parallel, their independence assumption causes acceptance rates to degrade sharply at later draft positions, limiting their scalability to large speculative budgets. In this paper, we analyze this limitation and introduce *Weaver*, a lightweight autoregressive adapter that constructs proposal trees from the top-K predictions of a factorized drafter. Weaver scales to large tree budgets by restoring conditional dependencies while avoiding the cost of a full-vocabulary projection. To verify these trees efficiently in recent models with Gated DeltaNet layers, we derive a rollback-free tree-verification algorithm and implement GPU kernels in CUDA and Metal. By combining these model and systems contributions, we achieve average lossless speedups of $4.09\times$ over autoregressive decoding and $1.22\times$ over a DFlash baseline on NVIDIA B200 with Qwen3.6-27B. For local deployment with the same model, we achieve average speedups over autoregressive decoding of $3.11\times$ on Apple M5 Max and $6.29\times$ on NVIDIA DGX Spark, with corresponding speedups of $1.35\times$ and $1.51\times$ over DFlash.

1 INTRODUCTION

On-device applications of autoregressive large language models (LLMs) require fast, energy-efficient token generation for individual requests. Recent open-weight models such as Qwen3.6-27B (Qwen Team, 2026b) offer strong reasoning and coding capabilities while remaining feasible to deploy locally on laptops with quantization. However, these deployments typically serve a single request at a time and therefore rarely benefit from batching, leaving the available compute underutilized due to the sequential nature of autoregressive decoding.

Speculative decoding (Leviathan et al., 2023) addresses this limitation by introducing parallelism within a single request. A fast auxiliary draft model generates a sequence of proposal tokens, and then compute their likelihood using the main model (called *verifier* hereafter) in parallel in a single step. An exact sampling procedure is then used to accept or reject each proposed token individually while preserving the verifier's output distribution.

Recently, factorized models inspired by discrete diffusion have been used to set new records in speculative decoding efficiency (An et al., 2025; Yu et al., 2026; An et al., 2026; Chen et al., 2026), as factorization enables fast parallel generation of proposals even with very large drafters. In single-request deployments, larger speculative budgets offer an opportunity to make further use of otherwise underutilized compute, as they can amortize the verifier's memory-access cost by evaluating more candidate tokens per step and increase the number of accepted tokens per step. Tree-based methods (Cai et al., 2024; Miao et al., 2024), exploit larger budgets to construct a prefix tree from

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.

multiple proposals. Unfortunately, marginal token distributions from factorized models diverge from true autoregressive conditionals as draft length increases, which puts hard limits on acceptance rates of factorized drafters, and makes them scale poorly with increased draft lengths, as we show in Figure 1.

A separate obstacle arises in hybrid models such as Qwen3.6-27B, which combine attention with Gated DeltaNet (GDN) layers (Qwen Team, 2026b; Yang et al., 2025). Tree verification for these recurrent layers must account for branch-specific state transitions; modifying the attention mask alone is insufficient.

In this paper, we introduce **Trees from Marginals (TfM)**, a tree-based speculative decoding method designed for low-latency local inference. It constructs prefix-conditioned proposal trees from factorized priors and supports efficient verification for Gated DeltaNet layers. Our contributions are summarized as follows:

- We introduce **Weaver**, a lightweight adapter which achieves excellent scalability for tree constructions by combining the benefits of factorized and autoregressive drafters. We use the top-K marginal predictions from a pre-trained factorized DFlash[1] (Chen et al., 2026) model as a prior for a small autoregressive drafter. We constrain the candidate vocabulary at each speculative position to the top-K marginal tokens to avoid the expensive multiplication by the output projection matrix, keeping the autoregressive overhead to a minimum.
- We derive an efficient tree-verification algorithm and implement fused kernels for CUDA and Metal.

We implement our speculative decoding algorithm and fused tree-verification kernel in both SGLang (Zheng et al., 2024) and Uzu (Mirai Tech Inc., 2025), a highly optimized Metal-based LLM inference runtime for Apple Silicon. Evaluated across both datacenter and consumer GPUs, our SGLang implementation achieves a lossless speedup in interactivity (tokens/sec/sequence) of up to $1.22\times$ on a single NVIDIA B200 and up to $1.51\times$ on NVIDIA DGX Spark over an optimized DFlash baseline for Qwen3.6-27B. On Apple Silicon, our Uzu implementation achieves speedups of up to $1.27\times$ on M5 Pro and $1.35\times$ on M5 Max under the same quantization method.

2 Preliminaries and Related Work

Notation. Speculative decoding uses a smaller drafter p_{drafter} to propose multiple tokens for parallel evaluation by the verifier p_{verifier}. For tree-based speculation, the drafter proposes a tree of depth D under a speculative budget of B tokens. For chain-based speculation, $B = D$. In each speculation step, the drafter proposes B tokens in T_{draft}. Verification takes T_{verify} and yields γ accepted draft tokens plus the bonus token produced by the verifier. The expected number of tokens generated per step is therefore $\tau = \mathbb{E}[\gamma + 1] \in [1, D + 1]$. Following *Sequoia* (Chen et al., 2025), and assuming sequential drafting and verification, the speedup over autoregressive decoding is $\text{Speedup} \approx \frac{\tau T_{\text{AR}}}{T_{\text{draft}} + T_{\text{verify}}}$, where T_{AR} is the latency spent on the single token.

2.1 Drafter Architectures

One design axis for drafter architectures is how a multi-token proposal is produced.

Autoregressive drafters. They emit proposal tokens sequentially, each conditioned on those before it, following the chain rule and diverging from the verifier only through drafter error. However, limited parallelism and vocabulary projection costs make it difficult to scale up the drafter within a fixed budget. *Hydra* (Ankner et al., 2024), *EAGLE* (Li et al., 2024b), *EAGLE-2* (Li et al., 2024a) and *EAGLE-3* (Li et al., 2025) all train lightweight adapters that reuse the residual stream activations of the verifier model and share its output projection matrix. *Gemma4-MTP* (Lacombe & Grootendorst, 2026) reduces vocabulary projection cost in its E2B and E4B variants by restricting the output projection to selected token clusters.

[1]The proposed method is not constrained to DFlash; the model was chosen as the current state-of-the-art factorized drafter. In practice one can use our method on top of any other parallel drafter, e.g. *I-DLM* (Yu et al., 2026).

Factorized drafters. They instead predict future tokens in a single forward pass, drawing each from a marginal that is conditionally independent of the realized prefix, trading prefix dependence[2] for parallelism and scalability. *PARD* (An et al., 2025) and *PARD-2* (An et al., 2026) parallelize autoregressive drafters using mask token placeholders. *Medusa* (Cai et al., 2024) uses multiple prediction heads, while *I-DLM* (Yu et al., 2026) fine-tunes the verifier itself for parallel drafting. *DFlash* (Chen et al., 2026) uses a large transformer-based adapter that decodes an entire block in one pass using non-causal in-block attention over mask tokens, with verifier hidden states injected into the KV cache of the draft model.

Prefix-dependent refinement. A complementary class of methods combines parallel drafting with lightweight prefix-dependent refinement. *DSpark* (Cheng et al., 2026) introduces intra-block dependencies through a lightweight sequential correction head, while *Domino* (Huang et al., 2026) uses a GRU-based causal encoder to refine parallel drafter logits. *DFlash2* (Inco AI, 2026) augments DFlash's marginal logits with a lightweight bigram path selector that scores compatibility between adjacent top-16 candidates in parallel. We developed our method independently of these approaches, with the goal of extending prefix-dependent refinement to tree-structured drafting.

2.2 Tree-based speculative decoding methods

Tree-based speculative decoding methods (Cai et al., 2024; Miao et al., 2024) construct a prefix tree from multiple proposals. A simple modification to the causal attention mask (usually referred to as tree attention) allows efficient evaluation of the verifier likelihood of every token in the tree in parallel. By providing multiple candidate tokens at each draft position, the tree makes better use of larger speculative budgets. *DDTree* (Ringel & Romano, 2026) extends DFlash to tree-based speculation via best-first search, scoring each prefix by the product of their position-wise marginal probabilities.

Tree shape optimization. The structure of the draft tree has a major impact on the performance of speculative decoding. Early methods such as *SpecInfer* (Miao et al., 2024) and *EAGLE* (Li et al., 2024b) used heuristic-based static tree structures. *Sequoia* (Chen et al., 2025) proposed an optimization method to statically select an optimal tree shape for a given pair of drafter and verifier. *DySpec* (Xiong et al., 2024) dynamically constructs token trees using greedy expansion based on draft probabilities.

Tree verification methods. *SpecInfer* (Miao et al., 2024) uses recursive rejection sampling across multiple candidates, while *Traversal Verification* (Weng et al., 2025) extends the sequence-level verification principle of *Block Verification* (Sun et al., 2025) to token trees. However, their correctness guarantees rely on assumptions about candidate sampling that arbitrary node selection and pruning need not preserve. In this work, we consider only *naive sampling* (Miao et al., 2024) and *Gumbel-max coupling* (Daliri et al., 2025) for verification to simplify the discussion and leave the integration of these alternative verification algorithms with our tree construction procedure to future work. Both verification methods are *lossless* under our tree construction procedure, as node selection and pruning affect only how far verification proceeds without altering the verifier's output distribution.

Tree verification for recurrent models. An ancestor-restricted attention mask enables single-pass tree verification for attention layers (Miao et al., 2024), however, recurrent layers require branch-specific states. Early methods used chain verification (Wang et al., 2024; Wu et al., 2024). *STree* (Wu et al., 2025) supports tree verification for diagonal state-space recurrences through an ancestor-masked scan. However, this reduction does not directly apply to *Gated DeltaNet* (Yang et al., 2025), whose updates involve non-commuting matrices $I - \beta_t k_t k_t^\top$.

3 Trees from Marginals

We now present **Trees from Marginals**, a tree-based speculative decoding method built around **Weaver**, a 60M-parameter autoregressive Transformer (Vaswani et al., 2017) drafter, but focus its capacity on learning residual corrections to DFlash (Chen et al., 2026)'s marginal logits.

[2]We consider single-pass parallel prediction rather than iterative denoising.

3.1 WEAVER

Our preliminary experiments examine whether accurate marginal predictions are sufficient to sustain acceptance at later draft positions. Figure 1 shows that proposals based on marginals estimated directly from the verifier still exhibit declining token-wise acceptance under *Speculative sampling* (Leviathan et al., 2023; Chen et al., 2023). A marginal distribution averages over possible draft prefixes, whereas the verifier conditions on the particular prefix being evaluated. Experimental details are provided in Appendix B.2.

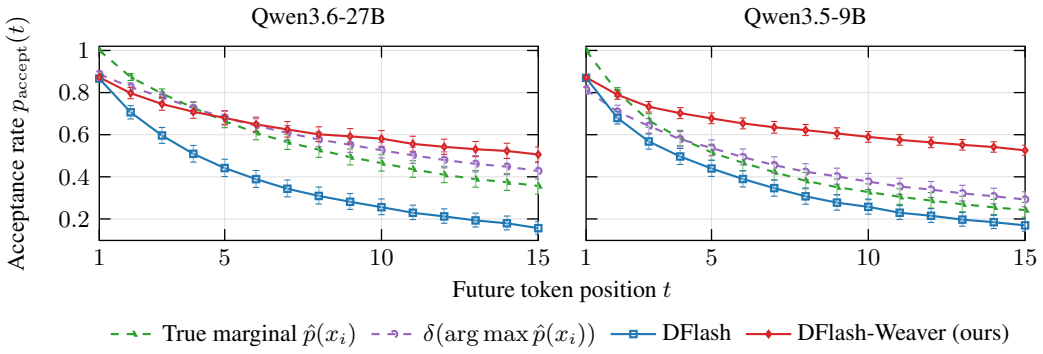


Figure 1: Acceptance rates under speculative sampling computed for different draft distributions. Error bars show 95% confidence intervals.

We also evaluate an autoregressive corrector that refines DFlash's distributions over the top-512 candidates[3] at each position, introducing causal dependencies into factorized drafting (the *DFlash-Weaver* curve). This corrector improves token-wise acceptance over DFlash and surpasses the estimated true-marginal baseline at later draft positions. Motivated by this experiment, we aim to retain the model capacity afforded by parallel drafting while adapting predictions to each tree prefix.

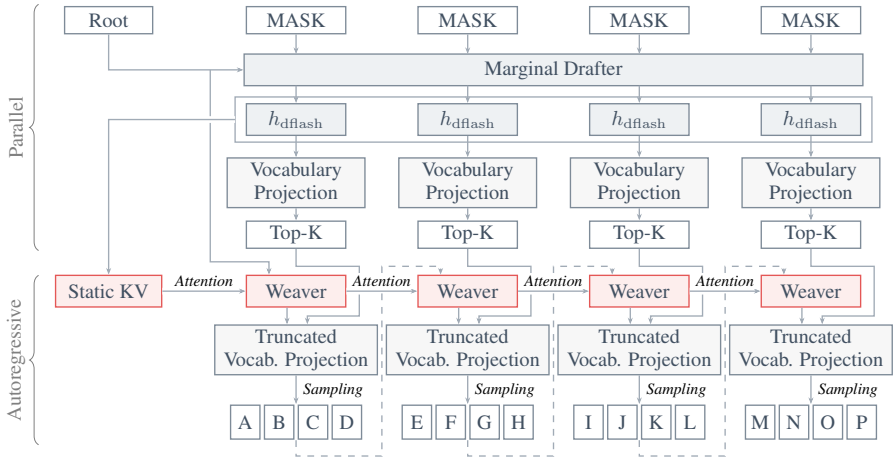


Figure 2: Weaver drafting procedure using a chain proposal for simplicity

Weaver (Figure 2) is a lightweight autoregressive transformer that learns prefix-dependent residual corrections to a factorized drafter's marginal logits to approximate the verifier's conditional distribution.

Factorized priors. At the start of a new speculative decoding round, DFlash takes in the last token verifier hidden state h_{verifier} to produce a sequence of future state lookaheads $h^1_{\text{dflash}}, \ldots, h^D_{\text{dflash}}$.

[3]We choose $K = 512$ so that the top-K candidates cover 93.8–99.9% of the verifier's tokens across draft positions (Appendix B.1).

For each drafting position $d \in [1, D]$, multiplying these states by the verifier vocabulary projection matrix W_{vocab} yields the baseline marginal logits in parallel: $\ell^d_{\text{dflash}} = W_{\text{vocab}} h^d_{\text{dflash}}$. In the original DFlash the tokens sampled using these logits *are* the draft tokens. We, however, select the top-512 token indices C_d and construct a truncated vocabulary projection matrix W'_d.

$$C_d = \text{TopK}(\ell^d_{\text{dflash}}, 512), W'_d = W_{\text{vocab}}[C_d, :]. \tag{1}$$

Context prefill. Weaver uses the verifier activations h_{verifier} and the DFlash lookahead states h^d_{dflash} to prefill a static KV cache shared across all tree branches. A conditioning weight matrix W_{c}, trained from scratch, projects the activations into continuous *prompt* tokens u: $u^0 = W_{\text{c}}\text{RMSNorm}(h_{\text{verifier}})$, $u^d = W_{\text{c}}\text{RMSNorm}(h^d_{\text{dflash}})$.

Autoregressive refinement. Finally, a small transformer WeaverStep uses the truncated vocabulary projection matrix W'_d to compute residual logits over C_d and the tokens along the draft tree path $t^{0..d}$. We apply *RoPE* (Su et al., 2023) to the attention queries and keys of both the continuous prompt tokens and subsequent draft tokens to encode their positions.

$$q^d_{\text{draft}} = \text{Softmax}\big(\ell^d_{\text{dflash}}[C_d] + \text{WeaverStep}(u^{0..D}, t^{0..d-1})\big)\,. \tag{2}$$

We use a single-layer transformer with a hidden size of 2048 and zero-initialize the final linear projection preceding the truncated vocabulary projection to stabilize training. We train approximately 60M Weaver parameters with the *LK loss* (Samarin et al., 2026) while keeping DFlash frozen; architecture and training details are provided in Appendix A.

3.2 Tree construction

T_{verify} dominates per step latency in local deployments. We therefore accept a modest increase in T_{draft} to verify only carefully selected candidates. We discuss how this trade-off pays off in Section 4.2. We extract up to w candidate nodes with the highest cumulative log-probabilities from the frontier, the set of expandable nodes, and use Weaver to add c children to each selected node in parallel. We repeat this procedure for R rounds, producing at most Rwc nodes. Finally, we retain the B nodes with the highest cumulative log-probabilities as the final proposal (Algorithm 1).

Algorithm 1 Pseudocode for our grow-and-prune tree construction algorithm

function BuildTree(r_t, B, w, c, R, D)
  input: root token r_t, budget B, batch width w, fan-out c, rounds R, maximum depth D
  root $\leftarrow$ Node(r_t, parent : $\varnothing, d : 0, s : 0$)
  Tree $\leftarrow$ {root}; Frontier $\leftarrow$ {root}
  for $i = 1$ **to** R **do**
    Batch $\leftarrow$ Frontier.PopTopExpandable(w, D)
    ▷ *Remove up to w nodes with largest path scores among nodes with $d < D$*
    if Batch $= \emptyset$ **then break**
    $\{q_n\}_{n \in \text{Batch}} \leftarrow$ WeaverStep(Batch)
    parallel for each $n \in$ Batch **do**
      for each $v \in$ SelectChildren(q_n, c) **do** ▷ *GMC: Gumbel Top-c (Shared seed); NS: Top-c.*
        $s_v \leftarrow n.s + \log q_n(v)$
        $u \leftarrow$ Node(v, parent : $n, d : n.d + 1, s : s_v$)
        Tree.Add(u); Frontier.Push(u)
  return PruneToBudget(Tree, B)

3.3 Tree verification for gated deltanet

We extend the chunked gated delta rule to tree-structured proposals, reducing the read-only verification step to an ancestor-masked triangular solve that we fuse into a single GPU kernel. A GDN

layer processes a sequence of tokens by maintaining a state matrix $S_t \in \mathbb{R}^{d_k \times d_v}$. At each step t, it takes a pair of key and value vectors $k_t \in \mathbb{R}^{d_k}$, $v_t \in \mathbb{R}^{d_v}$, as well as a decay factor $\alpha_t \in (0, 1]$ and a write strength factor $\beta_t \in (0, 1]$, and performs a state update:

$$S_t = \alpha_t (I - \beta_t k_t k_t^\top) S_{t-1} + \beta_t k_t v_t^\top \tag{3}$$

It then uses a query vector $q_t \in \mathbb{R}^{d_k}$ to extract the output o_t from the updated state:

$$o_t = \frac{q_t^\top S_t}{\sqrt{d_k}} \tag{4}$$

The chunk form of the delta rule allows to compute the result of n consecutive updates in a single pass via a pair of triangular solves (Yang et al., 2024) (Yang et al., 2025). We derive a simple extension of the chunked form which operates on trees of tokens. Let $\prec$ be an ancestor relation in a draft tree, then token i can only receive context from token j if $j \prec i$. We sort tokens in a draft tree in the topological order, such that $i \preceq j \Rightarrow i \leq j$ (Figure 3).

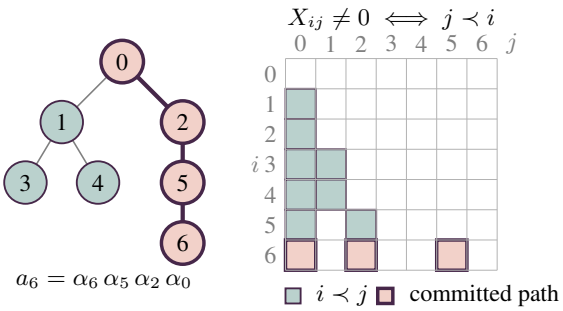


Figure 3: Tree verification modifies the GDN chunk algebra to use a partial order: a token at index i interacts only with its ancestors $j \prec i$.

Let matrices $K \in \mathbb{R}^{n \times d_k}$ and $V \in \mathbb{R}^{n \times d_v}$ be row-stacked key and value vectors, and let $a_t = \prod_{i \preceq t} \alpha_i$ be a sequence of cumulative products of decay factors along tree branches. The cumulative key-key and query-key interactions can be represented by lower-triangular matrices $X \in \mathbb{R}^{n \times n}$ and $Y \in \mathbb{R}^{n \times n}$ respectively:

$$X_{ij} = \mathbb{I}[j \prec i] \frac{a_i}{a_j} \beta_i k_i^\top k_j, \quad Y_{ij} = \mathbb{I}[j \preceq i] \frac{a_i}{a_j} q_i^\top k_j \tag{5}$$

Following Yang et al. (2025), we define auxiliary matrices W and U as solutions to linear systems

$$(I + X)U = \beta V, \quad (I + X)W = \beta a K \tag{6}$$

where $\beta = \mathrm{diag}(\beta_t)$ and $a = \mathrm{diag}(a_i)$. Then the outputs of the GDN layer in the form of a matrix $O \in \mathbb{R}^{n \times d_v}$ can be computed as

$$O = \frac{1}{\sqrt{d_k}} \left(a Q S_0 + Y (U - W S_0) \right) \tag{7}$$

The same quantities give the state at the last accepted node ℓ.

$$S_\ell = a_\ell S_0 + K^\top \mathrm{diag}\left(\mathbf{1}[j \preceq \ell] \frac{a_\ell}{a_j} \right) (U - W S_0), \tag{8}$$

where the indicator restricts contribution to ℓ and its ancestors. In practice, we replay the accepted path using Equation 3 at commit time. We describe our optimized CUDA kernel implementation in Appendix C.

4 EXPERIMENTS

Models and Evaluations. We evaluate our method across model sizes and architectures using **Llama3.1-8B** (Grattafiori et al., 2024), **Qwen3.5-9B** (Qwen Team, 2026a), **Qwen3.6-27B** (Qwen Team, 2026b), and **Muse Glimmer** (Meta Superintelligence Lab, 2026) as verifier models at batch size 1, over workloads that determine user-facing interactivity, in three categories: **Chat** (MT-Bench (Zheng et al., 2023b), ShareChat (Yan et al., 2026)), **Math** (GSM8K (Cobbe et al., 2021), MATH500 (Lightman et al., 2023), AIME25 (Zhang & Math-AI Team, 2025)), and **Code** (HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), LiveCodeBench (Jain et al., 2024)). All methods use the default sampling parameters of each verifier model. We use NVIDIA B200 to benchmark our method across models and datasets. Apple M5 Pro, M5 Max, and NVIDIA DGX Spark (GB10) to measure performance in local deployments. We evaluate two variants of TfM with different verification procedures: TfM-NS uses *Naive sampling* (Miao et al., 2024), while TfM-GMC uses *Gumbel-max coupling* (Daliri et al., 2025) for verification.

Baselines and Implementations. On NVIDIA B200 and DGX Spark, we compare our method with DFlash, DDTree and EAGLE-3, all implemented in SGLang (Zheng et al., 2024). On Apple M5 Pro and M5 Max, we compare with DFlash within Uzu (Mirai Tech Inc., 2025), an open-source inference runtime for Apple Silicon. We also compare against MTPLX (Altoukhi, 2026), an open-source inference runtime that uses native multi-token prediction (MTP) heads for speculative decoding on Apple Silicon. For all DFlash-based implementations, we use the same source drafter checkpoints listed in Appendix A.1. DDTree, EAGLE-3, and TfM use our optimized tree-verification kernels for GDN layers.

Metrics and Quantizations. We measure drafter quality using tokens/step (τ), and report lossless improvements in end-to-end interactivity (tokens/sec/sequence)[4] over the autoregressive baseline. In experiments on Apple M5 Pro and M5 Max, we evaluate **Qwen3.5-9B** and **Qwen3.6-27B** using Mirai-M quantization (Chakhvadze et al., 2026) and compare against autoregressive decoding of the same quantized models. For MTPLX, we also use the published quantized checkpoints (Youssofal, 2026a;b). For **Qwen3.6-27B** on NVIDIA DGX Spark, we use an FP8 checkpoint.

4.1 MAIN RESULTS

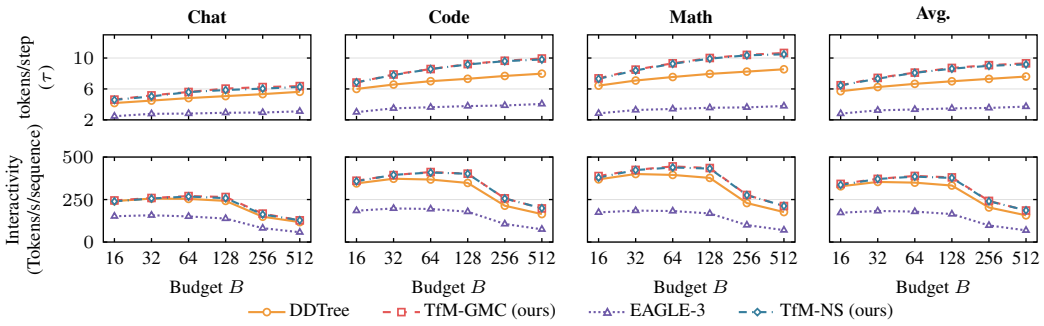


Figure 4: Acceptance length and interactivity across speculative budgets $B \in \{16, 32, 64, 128, 256, 512\}$ for DDTree, TfM, and EAGLE-3 on Qwen3.6-27B on B200.

Table 1 reports decoding speedups over autoregressive decoding on an NVIDIA B200. On this datacenter GPU, the gains in tokens per verification step do not offset the additional tree-construction latency for models with 9B parameters or fewer. For the two larger models, however, TfM outperforms both baselines across all three workload categories, with average speedups 22.1–24.5% higher than DFlash and 6.0–10.5% higher than DDTree. Across all settings, TfM achieves higher tokens/step than DDTree in the same speculative budget. Figure 4 reports tokens/step and cor-

[4]For each dataset, we aggregate $\tau = \sum \text{tokens} / \sum \text{steps}$ and $\text{Interactivity} = \sum \text{tokens} / \sum \text{time}$. Across datasets we use the unweighted mean.

Table 1: Decoding speedup ($\times$) over autoregressive decoding and average acceptance length (τ), with reasoning enabled for all models except Llama3.1-8B, and a maximum of 4096 new tokens. Parenthesized values indicate the speculation budget.

Target	Method		Chat				Math						Code						Avg.	
			MTBench		ShareChat		GSM8K		MATH500		AIME25		HumanEval		MBPP		LCB			
			$\times$	τ	$\times$	τ	$\times$	τ	$\times$	τ	$\times$	τ	$\times$	τ	$\times$	τ	$\times$	τ	$\times$	τ
Llama3.1 8B	DFlash	(10)	**3.01**	**4.08**	**2.33**	**3.12**	**3.20**	**4.46**	**3.73**	**5.01**	**4.20**	**5.57**	**3.69**	**5.06**	**3.50**	**4.89**	**3.14**	**4.25**	**3.35**	**4.56**
	DDTree	(64)	2.12	5.18	1.83	4.40	2.15	5.63	2.77	6.68	3.08	7.30	2.71	6.72	2.61	6.61	2.44	5.95	2.46	6.06
	EAGLE-3	(16)	1.90	4.25	1.12	2.43	1.86	4.40	1.54	3.32	1.43	3.03	2.04	4.54	2.03	4.67	1.92	4.24	1.73	3.86
	TfM-GMC	(64)	1.97	5.84	1.63	4.77	2.07	6.54	2.61	7.65	2.76	7.95	2.56	7.64	2.43	7.41	2.27	6.66	2.29	6.81
	TfM-NS	(64)	2.12	6.17	1.71	4.93	2.22	6.93	2.65	7.66	2.87	8.18	2.66	7.82	2.54	7.64	2.37	6.88	2.39	7.03
Qwen3.5 9B	DFlash	(16)	**1.96**	**3.37**	1.60	2.74	**2.44**	**4.20**	**3.04**	**5.24**	**2.93**	**5.06**	**2.82**	**4.94**	**2.89**	**5.04**	**2.24**	**3.88**	**2.49**	**4.31**
	DDTree	(64)	1.95	4.81	**1.62**	**3.98**	2.40	5.95	2.88	7.12	2.79	6.89	2.75	6.92	2.79	7.05	**2.24**	**5.56**	2.43	6.03
	TfM-GMC	(64)	1.90	5.58	1.59	4.64	2.38	7.02	2.82	8.33	2.72	7.99	2.64	7.85	2.71	8.14	2.21	6.52	2.37	7.01
	TfM-NS	(64)	1.88	5.38	1.55	4.44	2.38	6.90	2.85	8.28	2.75	7.96	2.69	7.87	2.73	8.05	2.21	6.39	2.38	6.91
Qwen3.6 27B	DFlash	(16)	2.60	3.80	2.01	2.93	3.74	5.54	4.09	5.98	3.79	5.53	3.84	5.64	3.73	5.46	2.99	4.36	3.35	4.90
	DDTree	(64)	2.96	5.31	2.40	4.29	4.02	7.33	4.40	7.90	4.13	7.39	4.19	7.54	4.09	7.36	3.40	6.11	3.70	6.65
	EAGLE-3	(32)	1.87	3.12	1.46	2.42	2.31	3.86	1.87	3.11	1.70	2.82	2.28	3.79	2.25	3.75	1.77	2.95	1.94	3.23
	TfM-GMC	(64)	**3.20**	**6.32**	**2.47**	**4.86**	**4.57**	**9.17**	4.89	9.71	4.58	9.09	4.68	9.30	4.53	9.01	3.78	7.48	**4.09**	**8.12**
	TfM-NS	(64)	**3.20**	**6.26**	2.46	4.81	4.51	9.00	**4.90**	**9.67**	**4.62**	**9.11**	**4.69**	**9.26**	**4.56**	**8.99**	**3.79**	**7.45**	**4.09**	**8.07**
Muse Glimmer (xhigh)	DFlash	(16)	2.22	3.11	2.02	2.83	3.01	4.22	3.15	4.42	2.98	4.17	2.85	3.98	2.81	3.92	2.48	3.47	2.69	3.77
	DDTree	(64)	2.70	4.63	2.47	4.24	3.48	6.01	3.61	6.24	3.44	5.95	3.34	5.74	3.32	5.70	2.93	5.09	3.16	5.45
	TfM-GMC	(64)	2.82	5.32	2.53	4.76	**3.74**	**7.08**	**3.86**	**7.29**	3.62	6.84	**3.60**	**6.76**	**3.55**	**6.67**	**3.10**	**5.88**	**3.35**	**6.33**
	TfM-NS	(64)	**2.83**	**5.30**	**2.54**	**4.75**	**3.74**	**7.07**	**3.86**	**7.28**	**3.65**	**6.90**	3.56	6.67	3.50	6.58	**3.10**	**5.86**	**3.35**	**6.30**

Table 2: Decoding speedup ($\times$) over standard and quantized autoregressive decoding at comparable quantization levels, with reasoning enabled. Generation is capped at 32768 new tokens on Apple M5 Pro and M5 Max, and 4096 on NVIDIA DGX Spark (GB10).

Hardware	Method		Qwen3.5-9B									Qwen3.6-27B								
			AR	MTBench		MATH500		HumanEval		**Avg.**		AR	MTBench		MATH500		HumanEval		**Avg.**	
			tok/s	$\times$	τ	$\times$	τ	$\times$	τ	$\times$	τ	tok/s	$\times$	τ	$\times$	τ	$\times$	τ	$\times$	τ
M5 Pro	MTPLX	(2 or 3)	28.31	2.25	3.15	2.36	3.05	2.64	3.01	2.42	3.07	11.54	2.70	3.03	3.03	3.53	2.98	3.41	2.90	3.32
	DFlash	(16)	52.69	1.74	3.74	2.50	5.22	2.53	4.75	2.26	4.57	17.91	2.53	3.87	3.44	5.66	3.25	5.57	3.07	5.04
	TfM-GMC	(16)		**2.27**	**5.03**	**2.85**	**6.69**	**3.04**	**6.40**	**2.72**	**6.04**		**3.24**	**5.23**	**4.28**	**7.51**	**4.16**	**7.42**	**3.89**	**6.72**
M5 Max	MTPLX	(2 or 3)	67.54	1.58	2.61	1.58	2.77	1.94	3.01	1.70	2.80	25.16	2.18	3.03	2.12	3.53	1.94	3.41	2.08	3.32
	DFlash	(16)	101.35	1.62	3.67	1.96	5.14	2.25	4.79	1.94	4.53	35.15	1.99	3.81	2.57	5.75	2.37	5.56	2.31	5.04
	TfM-GMC	(16)		**1.78**	**4.94**	**2.27**	**6.75**	**2.53**	**6.33**	**2.19**	**6.01**		**2.61**	**5.22**	**3.28**	**7.59**	**3.44**	**7.44**	**3.11**	**6.75**
DGX Spark	DFlash	(16)		2.70	3.64	4.06	5.48	4.06	5.47	3.61	4.86		3.10	3.94	4.93	6.28	4.47	5.70	4.17	5.31
	DDTree	(64)	13.31	3.24	5.24	4.54	7.36	4.27	6.91	4.01	6.51	7.70	4.20	5.50	6.20	8.14	5.83	7.66	5.41	7.10
	TfM-GMC	(64)		**3.57**	**6.05**	**5.24**	**8.89**	**5.05**	**8.55**	**4.62**	**7.83**		**4.62**	**6.30**	**7.36**	**10.07**	**6.89**	**9.40**	**6.29**	**8.59**

responding interactivity across different speculative budgets. TfM with $B = 32$ achieves higher average tokens/step than DDTree with $B = 256$ (7.43 vs. 7.30).

Table 2 also reports improvements in local deployment settings. On Apple M5 Pro and M5 Max, TfM achieves 12.9–34.6% higher average speedups than DFlash using the same runtime and quantization settings. On DGX Spark, TfM achieves average speedups of $4.62\times$ and $6.29\times$ over autoregressive decoding for Qwen3.5-9B and Qwen3.6-27B, respectively, exceeding DFlash by 28.0–50.8% and DDTree by 15.2–16.3%.

4.2 Drafting and verification latency

Table 3 decomposes per-round latency into parallel drafting, tree construction, and verification. Under the speedup model, the relative gain in τ must exceed the relative increase in total cycle latency. The reported latencies imply that outperforming DFlash requires approximately 31.4–64.5% higher τ on B200, compared with 5.7–10.2% on M5 Pro at the evaluated budgets. Against DDTree at the same budget of $B = 64$, the corresponding threshold is 8.3–20.6% on B200 and 4.1–6.3% on DGX

Table 3: Per-round latency breakdown (ms) into parallel drafting, tree construction, and verification on NVIDIA B200, Apple M5 Pro, and NVIDIA DGX Spark.

Hardware	Method	Qwen3.5-9B (ms/round)				Qwen3.6-27B (ms/round)			
		$T_{\text{draft}}(B)$		$T_{\text{verify}}(B)$	Sum	$T_{\text{draft}}(B)$		$T_{\text{verify}}(B)$	Sum
		DFlash	Tree			DFlash	Tree		
M5 Pro	DFlash (16)	11.53	-	43.44	54.97	12.27	-	112.78	125.05
	TfM-GMC (16)	12.14	5.43	43.04	60.60	12.64	5.46	114.09	132.18
B200	DFlash (16)	1.10	-	5.61	6.70	1.22	-	13.98	15.21
	DDTree (64)	1.49	0.43	7.22	9.14	1.62	0.44	16.39	18.45
	TfM-GMC (64)	1.50	2.19	7.32	11.02	1.61	2.15	16.23	19.99
DGX Spark	DFlash (16)	20.14	-	81.50	101.64	19.39	-	146.64	166.03
	DDTree (64)	19.82	0.65	100.47	120.93	19.38	0.65	151.15	171.19
	TfM-GMC (64)	20.42	7.56	100.57	128.55	19.38	7.50	151.38	178.26

Spark. The average τ gains of 32.2–33.3% over DFlash on M5 Pro and 20.3–21.0% over DDTree on DGX Spark exceed these estimated thresholds, consistent with the end-to-end speedups in Table 2.

4.3 Tree Verification Kernel Performance

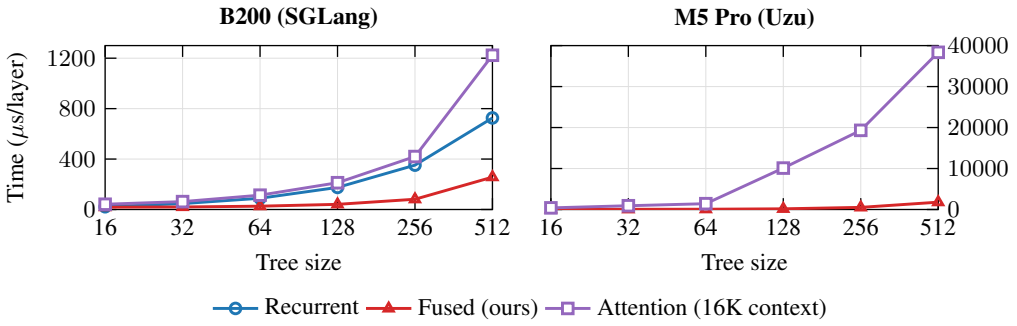


Figure 5: Per-layer tree forward pass time on B200 and M5 Pro, batch size 1, Qwen3.6-27B.

The fused GDN kernel avoids repeated state computations across tree branches, achieving a 1.17–4.33× speedup over the recurrent baseline on B200 (Figure 5). At a tree size of 128, per-layer latency decreases from 175.0 to 40.9 μs. On M5 Pro, the fused kernel takes 165.7 μs per layer at the same tree size; the attention reference with a 16K context takes 10.11 ms.

5 Conclusion

We investigated why factorized drafters stop scaling with the speculative budget and found that even a drafter using oracle marginal distributions exhibits declining acceptance at later positions. TfM uses a factorized drafter as a prior and adds Weaver, a 60M-parameter autoregressive adapter. Weaver learns prefix-dependent residual corrections over the top-K marginal candidates, avoiding full-vocabulary projections during tree expansion. We further derive an ancestor-masked chunked formulation of the gated delta rule and implement it as fused CUDA and Metal kernels. Local decoding benefits from allocating additional computation to prefix-dependent tree construction. Longer verification passes on M5 Pro and DGX Spark reduce the relative cost of Weaver's refinement, allowing the increase in tokens generated per cycle to outweigh the additional drafting latency and improve lossless decoding speed. Together, these components match the acceptance length of DDTree at one eighth of the budget and achieve average lossless speedups over autoregressive decoding of 3.11× on Apple M5 Max and 6.29× on NVIDIA DGX Spark with Qwen3.6-27B.

AI USE STATEMENT

We used generative AI tools to assist with code implementation and experiment execution. The authors designed the experiments and provided the specifications as prompts. The resulting code was reviewed by the authors for correctness before AI tools were used to execute the experiments. All experimental results were interpreted by the authors without AI assistance. The mathematical derivations for the GDN component were developed by the authors without AI assistance. We also used the verifier language model to generate synthetic training rollouts for Weaver, as described in the training procedure. The authors take responsibility for the final content of this work, including the AI-assisted implementation and experimental procedures.

REPRODUCIBILITY STATEMENT

Experimental settings and implementation details are described in Section 4 and Appendix A. The supplementary material includes the benchmark code, tree-verification GPU kernels, and code modifications used in our experiments. The supplementary `README` provides reproduction instructions and links to the trained drafter checkpoints and the quantized verifier checkpoints used in the Apple silicon experiments.

REFERENCES

Youssof Altoukhi. MTPLX. GitHub, 2026. URL https://github.com/youssofal/MTPLX.

Zihao An, Huajun Bai, Ziqiong Liu, Dong Li, and Emad Barsoum. PARD: Accelerating LLM Inference with Low-Cost PARallel Draft Model Adaptation. 4 2025. URL https://arxiv.org/abs/2504.18583v4.

Zihao An, Taichi Liu, Ziqiong Liu, Dong Li, Ruofeng Liu, and Emad Barsoum. PARD-2: Target-Aligned Parallel Draft Model for Dual-Mode Speculative Decoding. 5 2026. URL https://arxiv.org/abs/2605.08632v1.

Zachary Ankner, Rishab Parthasarathy, Aniruddha Nrusimha, Christopher Rinard, Jonathan Ragan-Kelley, and William Brandon. Hydra: Sequentially-Dependent Draft Heads for Medusa Decoding. 2 2024. URL https://arxiv.org/abs/2402.05109v2.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and others. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732*, 2021.

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. *Proceedings of Machine Learning Research*, 235:5209–5235, 1 2024. ISSN 26403498. URL https://arxiv.org/abs/2401.10774v3.

Artur Chakhvadze, Ryan Mathieu, Roman Knyazhitskiy, Nikolai Voinilenko, Chen-Chen Yeh, Artur Mullakhmetov, Eugene Bokhan, et al. Mirai Quantization: Redefining the speed-quality frontier for local llms on apple silicon, June 2026. URL https://trymirai.com/blog/quantization. In collaboration with others at Mirai Labs.

Sahil Chaudhary. Code Alpaca: An Instruction-following LLaMA model for code generation. https://github.com/sahil280114/codealpaca, 2023.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling, 2023. URL https://arxiv.org/abs/2302.01318.

Jian Chen, Yesheng Liang, and Zhijian Liu. DFlash: Block Diffusion for Flash Speculative Decoding. 2 2026. URL https://arxiv.org/abs/2602.06036v2.

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

Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. Sequoia: Scalable, robust, and hardware-aware speculative decoding, 2025. URL https://arxiv.org/abs/2402.12374.

Xin Cheng, Xingkai Yu, Chenze Shao, Jiashi Li, Yunfan Xiong, Yi Qian, Jiaqi Zhu, Shirong Ma, Xiaokang Zhang, Jiasheng Ye, Qinyu Chen, Chengqi Deng, Jiping Yu, Damai Dai, Zhengyan Zhang, Yixuan Wei, Yixuan Tan, Wenkai Yang, Runxin Xu, Yu Wu, Zhean Xu, Xuanyu Wang, Muyang Chen, Rui Tian, Xiao Bi, Zhewen Hao, Shaoyuan Chen, Huanqi Cao, Wentao Zhang, Anyi Xu, Huishuai Zhang, Dongyan Zhao, and Wenfeng Liang. Dspark: Confidence-scheduled speculative decoding with semi-autoregressive generation, 2026. URL https://arxiv.org/abs/2607.05147.

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. *arXiv preprint arXiv:2110.14168*, 2021.

Majid Daliri, Christopher Musco, and Ananda Theertha Suresh. Coupling without communication and drafter-invariant speculative decoding, 2025. URL https://arxiv.org/abs/2408.07978.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Jianuo Huang, Yaojie Zhang, Qituan Zhang, Hao Lin, Hanlin Xu, and Linfeng Zhang. Domino: Decoupling causal modeling from autoregressive drafting in speculative decoding, 2026. URL https://arxiv.org/abs/2605.29707.

Inco AI. DFlash 2: Keep Drafting Parallel, August 2026. URL https://inco.ai/blog/dflash2/.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

Olivier Lacombe and Maarten Grootendorst. Accelerating Gemma 4: Faster inference with multi-token prediction drafters, May 2026. URL https://blog.google/innovation-and-ai/technology/developers-tools/multi-token-prediction-gemma-4/.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 19274–19286. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/leviathan23a.html.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-2: Faster Inference of Language Models with Dynamic Draft Trees. *EMNLP 2024 - 2024 Conference on Empirical Methods*

in Natural Language Processing, Proceedings of the Conference, pp. 7421–7432, 6 2024a. doi: 10.18653/v1/2024.emnlp-main.422. URL https://arxiv.org/abs/2406.16858v2.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: Speculative Sampling Requires Rethinking Feature Uncertainty. *Proceedings of Machine Learning Research*, 235:28935–28948, 1 2024b. ISSN 26403498. URL https://arxiv.org/pdf/2401.15077.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-3: Scaling up Inference Acceleration of Large Language Models via Training-Time Test. 3 2025. URL https://arxiv.org/abs/2503.01840v3.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's Verify Step by Step. *arXiv preprint arXiv:2305.20050*, 2023.

Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, Yanru Chen, Huabin Zheng, Yibo Liu, Shaowei Liu, Bohong Yin, Weiran He, Han Zhu, Yuzhi Wang, Jianzhou Wang, Mengnan Dong, Zheng Zhang, Yongsheng Kang, Hao Zhang, Xinran Xu, Yutao Zhang, Yuxin Wu, Xinyu Zhou, and Zhilin Yang. Muon is Scalable for LLM Training. 2 2025. URL https://arxiv.org/pdf/2502.16982.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

Meta Superintelligence Lab. Muse Glimmer model card, August 2026. URL https://huggingface.co/meta-models/Muse-Glimmer-30B.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 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*, ASPLOS '24, pp. 932–949, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703867. doi: 10.1145/3620666.3651335. URL https://doi.org/10.1145/3620666.3651335.

Mirai Tech Inc. Uzu: A high-performance inference engine for AI models. GitHub, 2025. URL https://github.com/trymirai/uzu.

Dhruv Nathawani, Shuoyang Ding, Vitaly Lavrukhin, Igor Gitman, Somshubra Majumdar, Evelina Bakhturina, Boris Ginsburg, and Jane Polak Scowcroft. Nemotron-Post-Training-Dataset-v2, 8 2025. URL https://huggingface.co/datasets/nvidia/Nemotron-Post-Training-Dataset-v2.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026a. URL https://qwen.ai/blog?id=qwen3.5.

Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model, April 2026b. URL https://qwen.ai/blog?id=qwen3.6-27b.

Liran Ringel and Yaniv Romano. Accelerating Speculative Decoding with Block Diffusion Draft Trees. 4 2026. URL https://arxiv.org/abs/2604.12989v1.

Alexander Samarin, Sergei Krutikov, Anton Shevtsov, Sergei Skvortsov, Filipp Fisin, and Alexander Golubev. LK Losses: Direct Acceptance Rate Optimization for Speculative Decoding. 2 2026. URL https://arxiv.org/abs/2602.23881v2.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL https://arxiv.org/abs/2104.09864.

Ziteng Sun, Uri Mendlovic, Yaniv Leviathan, Asaf Aharoni, Jae Hun Ro, Ahmad Beirami, and Ananda Theertha Suresh. Block Verification Accelerates Speculative Decoding. 4 2025. URL http://arxiv.org/abs/2403.10444.

Teknium. OpenHermes 2.5: An Open Dataset of Synthetic Data for Generalist LLM Assistants, 2023. URL https://huggingface.co/datasets/teknium/OpenHermes-2.5.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Junxiong Wang, Daniele Paliotta, Avner May, Alexander M. Rush, and Tri Dao. The Mamba in the Llama: Distilling and Accelerating Hybrid Models. *Advances in Neural Information Processing Systems*, 37, 8 2024. ISSN 10495258. doi: 10.52202/079017-1996. URL https://arxiv.org/abs/2408.15237v4.

Yepeng Weng, Qiao Hu, Xujie Chen, Li Liu, Dianwen Mei, Huishi Qiu, Jiang Tian, and Zhongchao Shi. Traversal verification for speculative tree decoding, 2025. URL https://arxiv.org/abs/2505.12398.

Yangchao Wu, Yonatan Dukler, Matthew Trager, Alessandro Achille, Wei Xia, and Stefano Soatto. Snakes and Ladders: Accelerating SSM Inference with Speculative Decoding. In Mehdi Rezagholizadeh, Peyman Passban, Soheila Samiee, Vahid Partovi Nia, Yu Cheng, Yue Deng, Qun Liu, and Boxing Chen (eds.), *Proceedings of The 4th NeurIPS Efficient Natural Language and Speech Processing Workshop*, volume 262 of *Proceedings of Machine Learning Research*, pp. 292–304. PMLR, 7 2024. URL https://proceedings.mlr.press/v262/wu24a.html.

Yangchao Wu, Zongyue Qin, Alex Wong, and Stefano Soatto. STree: Speculative Tree Decoding for Hybrid State-Space Models. 5 2025. URL https://arxiv.org/abs/2505.14969v2.

Yunfan Xiong, Ruoyu Zhang, Yanzeng Li, Tianhao Wu, and Lei Zou. DySpec: Faster Speculative Decoding with Dynamic Token Tree Structure. 10 2024. URL https://arxiv.org/abs/2410.11744v1.

Yueru Yan, Tuc Nguyen, Bo Su, Melissa Lieffers, and Thai Le. ShareChat: A Dataset of Chatbot Conversations in the Wild, 2026. URL https://arxiv.org/abs/2512.17843.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing Linear Transformers with the Delta Rule over Sequence Length. *Advances in Neural Information Processing Systems*, 37, 6 2024. ISSN 10495258. doi: 10.52202/079017-3668. URL https://arxiv.org/pdf/2406.06484.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule, 2025. URL https://arxiv.org/abs/2412.06464.

Youssofal. Qwen3.6-27B-MTPLX-Optimized-Speed-V2. Hugging Face model repository, 2026a. URL https://huggingface.co/Youssofal/Qwen3.6-27B-MTPLX-Optimized-Speed-V2.

Youssofal. Qwen3.5-9B-MTPLX-Optimized-Speed. Hugging Face model repository, 2026b. URL https://huggingface.co/Youssofal/Qwen3.5-9B-MTPLX-Optimized-Speed.

Yifan Yu, Yuqing Jian, Junxiong Wang, Zhongzhu Zhou, Donglin Zhuang, Xinyu Fang, Sri Yanamandra, Xiaoxia Wu, Qingyang Wu, Shuaiwen Leon Song, Tri Dao, Ben Athiwaratkun, James Zou, Fan Lai, and Chenfeng Xu. Introspective Diffusion Language Models. 4 2026. URL https://arxiv.org/abs/2604.11035v1.

Yifan Zhang and Math-AI Team. American invitational mathematics examination (aime) 2025. https://huggingface.co/datasets/math-ai/aime25, 2025. Hugging Face dataset.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric. P Xing, Joseph E Gonzalez, Ion Stoica, and Hao Zhang. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset, 2023a.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023b. URL https://arxiv.org/abs/2306.05685.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, 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 A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdf.

A TRAINING DETAILS

A.1 CHECKPOINT DETAILS

For each verifier, we train Weaver using the corresponding publicly released DFlash checkpoint. The same source drafter checkpoint is used for Weaver training and all DFlash and DDTree baselines. Table 4 lists the checkpoint repositories and revisions, including those used for the EAGLE-3 experiments.

Table 4: Source drafter checkpoints used for training and evaluation.

Verifier	Drafter checkpoint	Revision
Llama3.1-8B	`Anbeeld/LLaMA3.1-8B-Instruct-DFlash-UltraChat-GGUF`	`bde23d8`
Llama3.1-8B	`yuhuili/EAGLE3-LLaMA3.1-Instruct-8B`	`ada412b`
Qwen3.5-9B	`z-lab/Qwen3.5-9B-DFlash`	`5fc3b3d`
Qwen3.6-27B	`z-lab/Qwen3.6-27B-DFlash`	`0919688`
Qwen3.6-27B	`Ex0bit/Qwen3.6-27B-PRISM-EAGLE3/compressed`	`5bf3915`
Muse Glimmer	`meta-models/Muse-Glimmer-30B-assistant`	`e8192f3`

A.2 TRAINING IMPLEMENTATION

We generate offline verifier rollouts with default sampling parameters, reasoning enabled, and a maximum response length of 8192 tokens. We store only token IDs and recompute verifier logits and DFlash lookahead states during training. Following DFlash (Chen et al., 2026), we resample 128 anchor positions per sequence, treating each anchor token as the verified bonus token that conditions drafting at inference.

At each lookahead position $j \in \{1, \dots, D\}$ following an anchor, Weaver predicts residual logits only for the DFlash candidate set C_j of size $K = 512$. We therefore restrict the verifier and proposal distributions to C_j and renormalize them, denoting the resulting distributions by p and q, respectively. Computing the teacher logits then requires only K dot products with the corresponding rows of the frozen vocabulary projection.

We train Weaver using the LK loss (Samarin et al., 2026),

$$\mathrm{L}_{\mathrm{LK}}^{\lambda}(p, q) = \lambda \ \mathrm{KL}(p \| q) + (1 - \lambda) \ \mathrm{TV}(p, q), \tag{9}$$

with the mixing weight computed independently at each position as

$$\lambda = \exp\big(-\eta \ \mathrm{stop_gradient}\big(1 - \mathrm{TV}(p, q)\big)\big) .$$

The loss emphasizes forward KL when the distributions are far apart and shifts weight toward TV as they become closer. We set $\eta = 2$.

For all verifiers, Weaver has 60.9M trainable parameters and uses a single Transformer layer with hidden dimension 2048, 16 attention heads, rotary positional embeddings (RoPE), and a SwiGLU MLP with intermediate dimension 2048. We train the large weight matrices with Muon (Liu et al., 2025) and the remaining parameters with AdamW (Loshchilov & Hutter, 2019), under a warmup–stable–decay (WSD) schedule with a peak learning rate of 2×10^{-4}.

We train Weaver on the Nemotron Post-Training Dataset V2 (Nathawani et al., 2025), LMSYS-Chat-1M (Zheng et al., 2023a), OpenHermes 2.5 (Teknium, 2023), and CodeAlpaca (Chaudhary, 2023). We subsample the union to 300k completions and train for a single epoch.

A.3 REASONING EFFORT

For models with multiple reasoning-effort settings, such as Muse Glimmer, we generate training rollouts by sampling the reasoning effort uniformly at random from the available settings. Table 5 reports benchmark results for Muse Glimmer at low, medium, and high reasoning effort.

Table 5: Muse Glimmer at low, medium, and high reasoning effort on a single NVIDIA B200

Effort	Method		Chat				Math						Code						Avg.	
			MTBench		ShareChat		GSM8K		MATH500		AIME25		HumanEval		MBPP		LCB			
			Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ
low	DFlash	(16)	2.20×	3.10	1.99×	2.76	3.82×	5.44	3.60×	5.04	3.18×	4.39	3.82×	5.39	3.78×	5.32	2.52×	3.49	3.11×	4.37
	DDTree	(64)	2.69×	4.60	**2.38×**	**4.06**	4.08×	7.36	4.05×	7.00	3.71×	6.32	4.17×	7.32	4.15×	7.26	3.01×	5.14	3.53×	6.13
	TfM-GMC	(64)	2.69×	5.07	2.35×	4.40	4.32×	8.53	4.17×	7.95	**3.84×**	**7.21**	4.29×	8.29	4.31×	8.31	**3.17×**	**5.96**	3.64×	6.96
	TfM-NS	(64)	**2.70×**	**5.07**	2.36×	4.40	**4.36×**	**8.66**	**4.23×**	**8.00**	3.82×	7.11	**4.35×**	**8.38**	**4.32×**	**8.26**	3.14×	5.88	**3.66×**	**6.97**
medium	DFlash	(16)	2.26×	3.15	2.01×	2.80	3.31×	4.64	3.35×	4.68	3.08×	4.27	3.25×	4.54	3.28×	4.58	2.46×	3.42	2.87×	4.01
	DDTree	(64)	2.70×	4.64	2.42×	4.14	3.68×	6.43	3.78×	6.52	3.45×	5.94	3.75×	6.50	3.73×	6.46	2.92×	5.05	3.30×	5.71
	TfM-GMC	(64)	2.72×	5.18	**2.44×**	**4.58**	**3.95×**	**7.59**	**4.05×**	**7.68**	3.65×	6.90	**3.96×**	**7.53**	**3.94×**	**7.48**	**3.06×**	**5.80**	**3.47×**	**6.59**
	TfM-NS	(64)	**2.77×**	**5.18**	**2.44×**	**4.55**	3.92×	7.52	**4.05×**	**7.65**	**3.71×**	**6.99**	3.94×	7.45	3.92×	7.41	3.03×	5.71	**3.47×**	**6.56**
high	DFlash	(16)	2.26×	3.15	2.04×	2.83	3.08×	4.29	3.23×	4.49	3.01×	4.17	2.93×	4.08	2.90×	4.03	2.48×	3.44	2.74×	3.81
	DDTree	(64)	2.72×	4.63	2.47×	4.20	3.51×	6.05	3.70×	6.34	3.51×	6.02	3.46×	5.92	3.42×	5.84	2.94×	5.07	3.22×	5.51
	TfM-GMC	(64)	**2.82×**	**5.29**	2.52×	4.72	**3.80×**	**7.21**	**3.91×**	**7.37**	**3.66×**	**6.90**	**3.71×**	**6.96**	**3.63×**	**6.83**	**3.12×**	**5.90**	**3.40×**	**6.40**
	TfM-NS	(64)	2.81×	5.24	**2.53×**	**4.71**	**3.80×**	**7.15**	**3.91×**	**7.33**	3.64×	6.84	3.68×	6.86	**3.63×**	**6.78**	3.08×	5.80	3.38×	6.34

B Ablation Study

B.1 The effect of truncated vocabulary projection

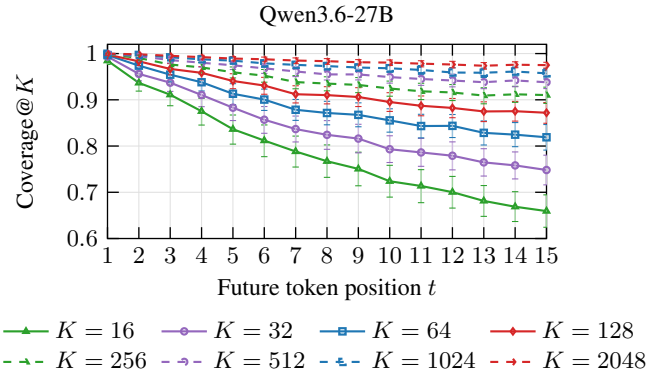


Figure 6: Verifier-token coverage of DFlash Top-K candidates, estimated using 128 continuations per anchor under Gumbel-max coupling.

We evaluate candidate coverage using Qwen3.6-27B as the verifier (Figure 6). For each completion, we randomly select eight anchor positions and generate 128 independent continuations of 15 tokens per anchor. Let p_d denote the verifier distribution conditioned on the sampled prefix and q_d the DFlash marginal at position d, computed once at the anchor. We sample both distributions over the full vocabulary using shared Gumbel noise:

$$Y_d^r = \arg\max_{v \in \mathcal{V}} \big[\log r_d(v) + G_{d,v}\big], \qquad r \in \{p, q\}, \quad G_{d,v} \overset{\text{i.i.d.}}{\sim} \text{Gumbel}(0, 1).$$

For the candidate set $C_{d,K} = \text{TopK}(q_d, K)$, verifier coverage is

$$\text{Coverage@K}(d) = \mathbb{E}[\mathbf{1}\{Y_d^p \in C_{d,K}\}] = \mathbb{E}\left[\sum_{v \in C_{d,K}} p_d(v)\right], \tag{10}$$

where the expectation is over anchors, verifier continuations, and Gumbel noise. Figure 6 reports empirical coverage at each draft position for $K \in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$, using the same samples for all values of K.

B.2 Empirical acceptance of factorized drafting

For the simplicity of analysis, we consider only speculative sampling. During chain speculative sampling the token-wise acceptance probability is exactly $1 - \mathrm{D}_{\mathrm{TV}}(p_{\mathrm{draft}}, p_{\mathrm{verifier}})$. For a future position $L + t$ where L is the index of the last verified token, an oracle marginal drafter would use the true future-token marginal[5]

$$p(x_{L+t} \mid x_{\leq L}) = \mathbb{E}\left[p_{\mathrm{verifier}}(x_{L+t} \mid x_{<L+t}) \mid x_{\leq L}\right] \tag{11}$$

We estimate these marginal probabilities and evaluate the resulting oracle-marginal baseline. For each context sequence of length $L < 4096$, we sample $M = 128$ random continuations x^m from the verifier. We then run the verifier on the produced token sequence and compute the output token distribution $p(x_{L+t} \mid x^m_{<L+t})$ at each position.

We then estimate the position-wise marginal probabilities as

$$\hat{p}(x_{L+t} \mid x_{\leq L}) = \frac{1}{M} \sum_{m=1}^{M} p(x_{L+t} \mid x_{\leq L}, x^m_{<L+t}). \tag{12}$$

The acceptance rate at a future position t can be then estimated as

$$p_{\mathrm{accept}}(t) = 1 - \mathbb{E}\left[\mathrm{D}_{\mathrm{TV}}\left(\hat{p}(\cdot \mid x_{\leq L}), p(\cdot \mid x_{\leq L}, x^m_{<L+t})\right) \mid x_{\leq L}\right] \tag{13}$$

where expectation is taken over the context sequences and continuations in the dataset.

We evaluate these proposals on MTBench (Zheng et al., 2023b). Figure 1 shows that proposals based on estimated verifier marginals still exhibit declining acceptance at later positions. Weaver achieves higher measured acceptance at later positions than both the oracle-marginal and argmax-marginal baselines evaluated here. These results motivate adapting proposal distributions to the realized prefix. We evaluate tree-budget scaling separately in Figure 4.

At later positions, the deterministic argmax-marginal proposal achieves higher measured token-wise acceptance than sampling from the estimated marginal, illustrating that marginal matching need not maximize acceptance. This observation is consistent with the advantage of naive verification with argmax drafting over speculative sampling for DFlash in Table 6. It also motivates position-dependent temperature scaling, which we examine alongside prefix conditioning in Appendix B.4.

B.3 Weaver on chain speculation

We compare two configurations for chain speculation: argmax proposals with naive sampling, and categorical proposals with speculative sampling. We evaluate both configurations for DFlash and TfM on Qwen3.6-27B.

Table 6: Decoding speedup over standard autoregressive decoding and (τ) on MTBench dataset with Qwen3.6-27B in bfloat16 at temperature 1.0 and reasoning enabled.

Method	Proposal	Verification Method	Speedup	τ
DFlash	$\mathrm{argmax}(P_{\mathrm{marg}})$	Naive Sampling	**2.67**×	**4.44**
TfM (Chain)	$\mathrm{argmax}(P_{\mathrm{AR}})$	Naive Sampling	2.42×	4.90
DFlash	$x \sim P_{\mathrm{marg}}$	Speculative Sampling	2.37×	4.06
TfM (Chain)	$x \sim P_{\mathrm{AR}}$	Speculative Sampling	**2.48**×	**5.00**

For DFlash, argmax proposals with naive sampling yield higher τ and decoding speedup than categorical proposals with speculative sampling (Table 6).

[5]The true marginal minimizes the expected forward KL divergence from the verifier conditionals, but need not minimize expected TV or maximize acceptance.

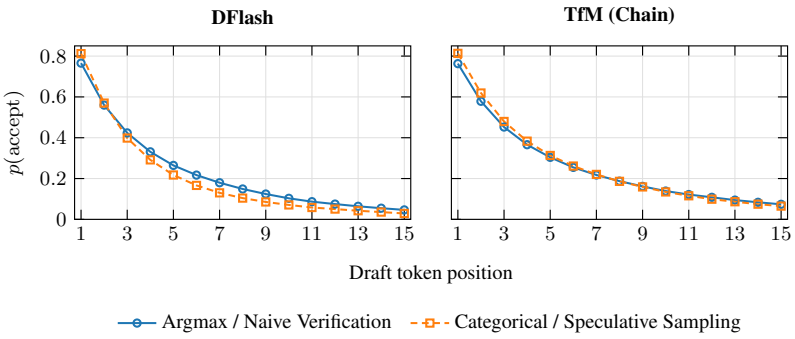


Figure 7: DFlash and TfM: acceptance by draft token position on MTBench with temperature 1.0

For DFlash, the acceptance curves cross between draft positions 2 and 3 (Figure 7). Categorical proposals with speculative sampling achieve higher acceptance at the first two positions, while argmax proposals with naive sampling achieve higher acceptance thereafter. Since argmax drafting corresponds to temperature-zero sampling, this crossover motivates annealing the draft temperature with depth, retaining stochasticity at early positions and sharpening the proposal distribution at later positions. We investigate this motivation in Appendix B.4 by selecting the draft temperature independently at each position.

B.4 Decomposing the acceptance gain

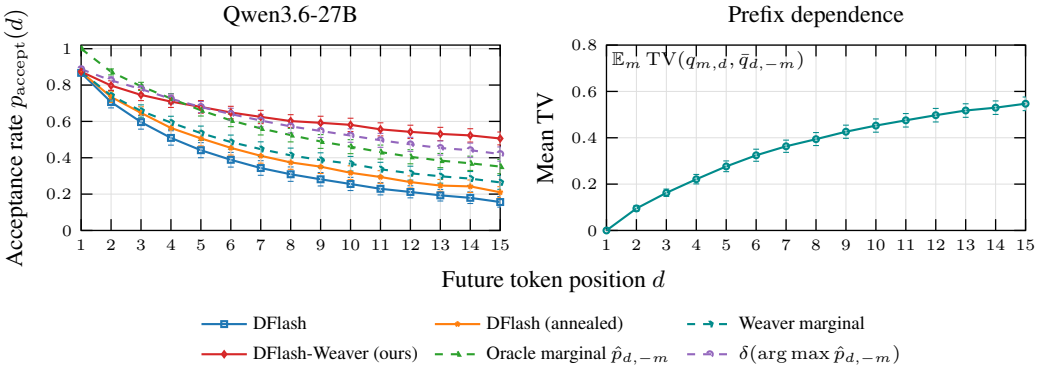


Figure 8: Acceptance and prefix dependence on Qwen3.6-27B. Left shows token-wise acceptance under speculative sampling. Right shows mean TV between conditional Weaver and its leave-one-out marginal. Error bars show pointwise 95% confidence intervals.

We evaluate Qwen3.6-27B on the 80 first-turn MTBench questions, using 16 anchors per question and $M = 128$ verifier continuations per anchor. All methods use the same anchors and continuations. For a fixed anchor, let $p_{m,d}$ and $q_{m,d}$ denote the verifier and conditional Weaver distributions at future position d along verifier continuation m. Table 7 summarizes the proposals compared in this experiment. Prefix dependence refers to conditioning on the realized continuation after the anchor; all proposals condition on the anchor context.

Here, $r_d(T) = \mathrm{softmax}(\ell_d^{\mathrm{DF}}/T)$ is the DFlash distribution at temperature $T > 0$, where ℓ_d^{DF} denotes its logits, and $r_d(0)$ denotes its $T \to 0$ limit, the deterministic DFlash argmax proposal. The temperature T_d is selected independently at each position by question-level cross-fitting. We use five folds, selecting temperatures on 64 questions and evaluating them on the remaining 16 in each fold.

The leave-one-out marginals average normalized probabilities over the other verifier continuations from the same anchor:

Table 7: Proposal distributions used in the acceptance comparison. All marginal estimates exclude the evaluated continuation m.

Method	Proposal	Prefix dependence
DFlash	$r_d(1)$	No
DFlash (annealed)	$r_d(T_d)$	No
Weaver marginal	$\bar{q}_{d,-m}$	No
DFlash-Weaver (ours)	$q_{m,d}$	Yes
Oracle marginal	$\hat{p}_{d,-m}$	No
Oracle marginal argmax	$\delta_{\mathrm{argmax}_v \hat{p}_{d,-m}(v)}$	No

$$\bar{q}_{d,-m} = \frac{1}{M-1} \sum_{j \neq m} q_{j,d}, \qquad \hat{p}_{d,-m} = \frac{1}{M-1} \sum_{j \neq m} p_{j,d}. \tag{14}$$

Marginalizing Weaver holds its parameters and candidate sets fixed while removing dependence on the evaluated continuation. Position-averaged acceptance is 37.79% for DFlash, 46.69% for the Weaver marginal, and 63.43% for conditional Weaver. Marginalization therefore removes approximately 65% of Weaver's acceptance gain over DFlash. Most of the observed gain thus depends on autoregressive prefix conditioning and cannot be explained by increased parameter count alone.

B.5 The effect of Weaver on tree speculation

Table 8: Weaver ablation and DDTree comparison on MTBench with Qwen3.6-27B. All methods use SGLang on a single NVIDIA B200 with $B = 64$.

Method	Sampling	τ	Tokens/s/seq	Rel. DDTree	T_{tree} (ms)
DDTree	Naive	5.31 $\pm$0.04	278.9 $\pm$2.3	1.00 $\pm$0.00	0.435
DDTree (Top-512 norm.)	Naive	5.41 $\pm$0.02	282.1 $\pm$0.7	1.01 $\pm$0.01	0.533
Algorithm 1, Weaver OFF	Naive	5.25 $\pm$0.06	266.0 $\pm$2.8	0.95 $\pm$0.02	1.048
Algorithm 1, Weaver ON	Naive	6.26 $\pm$0.06	300.5 $\pm$3.9	1.08 $\pm$0.01	2.201
Algorithm 1, Weaver OFF	GMC	5.32 $\pm$0.02	269.9 $\pm$1.3	0.97 $\pm$0.01	1.062
Algorithm 1, Weaver ON	GMC	6.32 $\pm$0.02	301.7 $\pm$1.1	1.08 $\pm$0.01	2.202

Table 9: Weaver ablation and DDTree comparison on MTBench with Qwen3.6-27B. All methods use SGLang on a single NVIDIA B200 with $B = 16$.

Method	Sampling	τ	Tokens/s/seq	Rel. DDTree	T_{tree} (ms)
DDTree	Naive	4.61 $\pm$0.01	264.5 $\pm$1.6	1.00 $\pm$0.00	0.378
DDTree (Top-512 norm.)	Naive	4.58 $\pm$0.02	261.4 $\pm$1.0	0.99 $\pm$0.00	0.472
Algorithm 1, Weaver OFF	Naive	4.61 $\pm$0.03	255.1 $\pm$2.0	0.96 $\pm$0.01	1.047
Algorithm 1, Weaver ON	Naive	5.13 $\pm$0.02	266.1 $\pm$3.0	1.01 $\pm$0.01	2.213
Algorithm 1, Weaver OFF	GMC	4.64 $\pm$0.03	256.0 $\pm$1.7	0.97 $\pm$0.01	1.062
Algorithm 1, Weaver ON	GMC	5.14 $\pm$0.00	267.3 $\pm$1.4	1.01 $\pm$0.00	2.226

Table 8 isolates Weaver's contribution by disabling its residual corrections while retaining the same tree-construction procedure. Without Weaver, Algorithm 1 achieves similar τ to DDTree but incurs higher tree-construction latency and lower throughput. Enabling Weaver increases τ from 5.25 to 6.26 under naive sampling and from 5.32 to 6.32 under GMC, yielding approximately 8% higher throughput than DDTree in both settings. Table 9 repeats the ablation with a tree budget of $B = 16$. Enabling Weaver increases τ from 4.61 to 5.13 under Naive sampling and from 4.64 to 5.14 under GMC.

B.6 THE EFFECT OF TREE SHAPE

We study the sensitivity of Algorithm 1 to its tree-construction parameters using Qwen3.6-27B on a single NVIDIA B200. We fix the number of expansion rounds to $R = 16$ and vary the expansion width w and fan-out c. We evaluate these configurations on MTBench and on a development subset of 80 prompts sampled from the training corpus, using the latter for configuration selection.

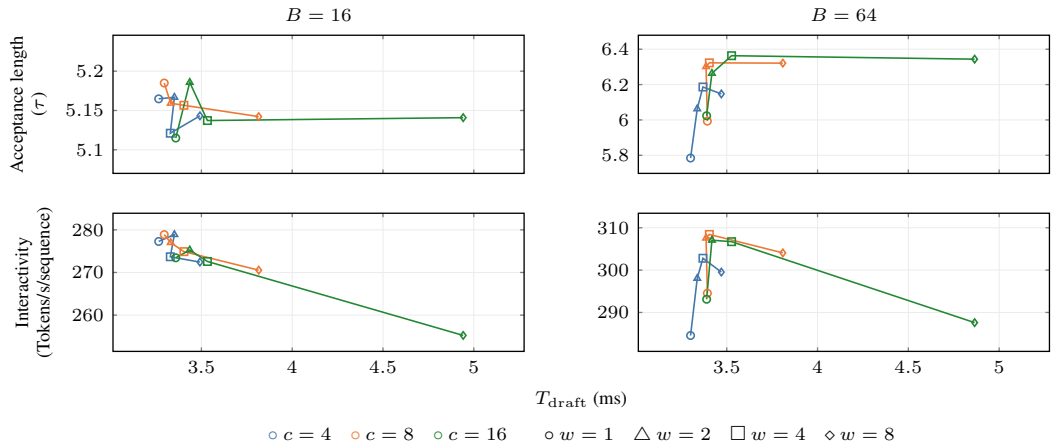


Figure 9: Tree-construction parameter sweep on MTBench with Qwen3.6-27B on B200 under shared Gumbel coupling. We fix $R = 16$ and vary expansion width w and fan-out c (color).

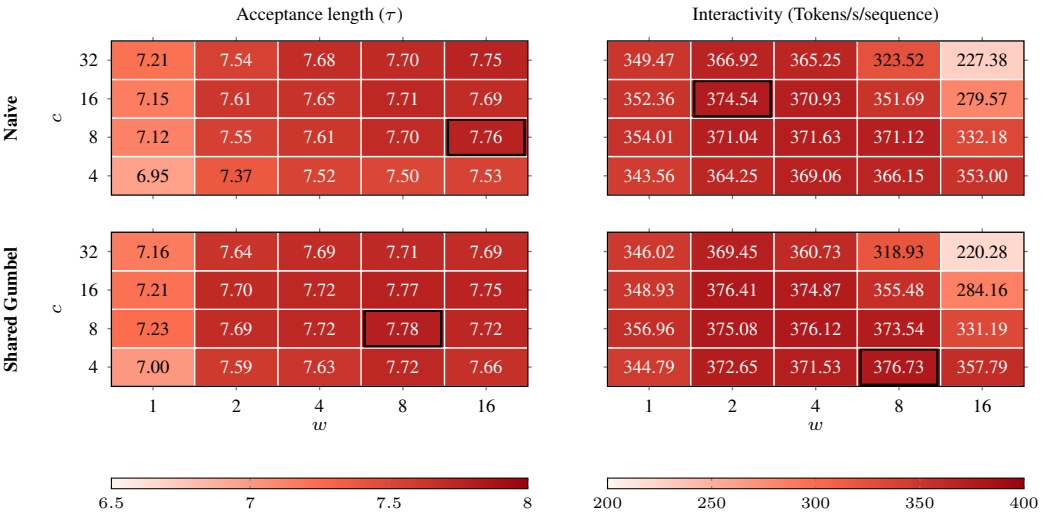


Figure 10: Tree-construction parameter sweep on 80 development prompts with Qwen3.6-27B on B200 under NS and shared Gumbel coupling. We fix $R = 16$; results are averaged over three seeds.

Figure 9 shows that, for $B = 64$, tokens per step range from 5.78 to 6.32 at comparable drafting latencies of approximately 3.3–3.4 ms, depending on the tree-construction parameters.

On the development subset (Figure 10), $(w, c) = (8, 8)$ achieves throughput within 1% of the best observed setting under both sampling schemes. Based on these results, we use $(R, w, c) = (16, 8, 8)$ for CUDA experiments. For Apple M5 Pro and M5 Max, we use the smaller configuration $(R, w, c) = (16, 4, 4)$.

C BATCHED FUSED-KERNEL TREE IMPLEMENTATION

The proposal tree is shared by every GDN layer, so its verification metadata is computed once per decode step. From the parent pointers we compute, for each node, the cumulative decay a_i (accumulated in log space) and the set of its proper ancestors; the ancestor sets are stored as bitmasks, one bit per candidate ancestor, packed into 64-bit words. The kernel inputs are dense tensors of shape $[N, T, H, d]$, with N the batch, T the padded tree size, and H the number of key heads. These shapes do not depend on the tree's contents, so the whole pass is captured once and replayed inside a CUDA graph. The kernel computes [scalars], [write strengths β and cumulative decays a, for all heads at once], [3.4], [4.9], builds the interaction matrix from the Equation 5 per key head, and runs the forward substitution tiled into sub-blocks of $B_c = 32$ nodes. Each diagonal block $(I + X_{bb})$ is inverted in registers by repeated squaring, and a cascade over sub-blocks handles the off-diagonal coupling,

$$U_b = (I + X_{bb})^{-1} \left((\beta \odot V)_b - \sum_{c<b} X_{bc} U_c \right), \tag{15}$$

which is $\lceil T/B_c \rceil$ dense block steps rather than T scalar steps. The $-w_j^\top S_0$ term of Equation 7 is folded into the right-hand side of the solve, so W is never formed and the substitution returns $u_j - w_j^\top S_0$ directly; the output stage then reads S_0 once, decayed, and applies the ancestor-masked attention against the solve result.

We lay the state out as $[H_v, d_v, d_k]$ to match the decode and prefill GDN kernels (H_v is the number of value heads). All matrix products use `tl.dot` with tf32 inputs and fp32 accumulation, except for the sensitive log-decay accumulations, which use tf32x3 ($\sim 5 \times 10^{-7}$ relative); the precision policy keeps the verify within 10^{-4} of the double-precision reference. Once tree verification picks the accepted leaf, a single commit pass replays Equation 3 along that path and writes S_0, which is the only state write in the entire decode step.

Table 10: Stages of the fused verify kernel and their self-time (µs, B200, batch size 1). The Gram and the solve dominate, and the solve is the one stage whose cost grows super-linearly with T (a factor of 2.1 from $T = 64$ to 128). The self-times sum to the compute portion; the rest of the Figure 5 wall-clock is kernel launch and inter-stage copies, fixed under CUDA-graph replay.

stage	computes	$T = 64$	$T = 128$
scalars	write strengths β and path sums G, for all heads at once	3.4	4.9
Gram	the interaction matrix X per key head, and the inverse of each diagonal sub-block	12.2	16.7
solve	the forward substitution (Equation 15)	6.5	13.8
output	the block output Equation 7	5.7	9.3
total		**27.8**	**44.7**