Custom Forward-Backward VJPs for DFA-Guided Diffusion Language Models: An Empirical Study

FARS
Analemma
fars@analemma.ai

Abstract

DFA-guided diffusion language models enable constrained text generation by steering denoising with gradients of DFA acceptance probability. However, the DFA dynamic programming computation accounts for 57–59% of each guided step, creating a significant bottleneck. We implement custom forward-backward vector-Jacobian products (VJPs) that analytically compute gradients without autograd tape storage, using Triton kernels and pre-allocated buffers. Our approach produces numerically identical gradients to baseline autograd (cosine similarity 1.0, relative L2 error 1.7×10^{-5}). However, we achieve only 1.01–1.23× speedup over `torch.compile`—far below our 3× target. The root cause is that tokenizer-aligned DFAs are inherently dense (50–6,177 edges per state-pair), invalidating sparse optimization approaches. We document this negative result to inform future work: accelerating DFA-guided diffusion likely requires alternative approaches such as state-space reduction or approximate inference rather than gradient computation optimizations.

WARNING: This paper was generated by an automated research system. The code is publicly available.[1]

1 Introduction

Diffusion language models have emerged as a promising alternative to autoregressive generation, offering advantages such as parallel token generation and continuous optimization landscapes (Ho et al., 2020; Li et al., 2022; Gulrajani & Hashimoto, 2023). Recent work has extended these models to support constrained generation through DFA-guided sampling, where the gradient of an analytically computed DFA acceptance probability steers the denoising process toward outputs satisfying regular constraints such as JSON schemas or regex patterns (Kim et al., 2026; Suresh et al., 2025).

However, DFA-guided diffusion suffers from substantial computational overhead. Profiling reveals that the DFA dynamic programming computation—computing acceptance probability and its gradient—accounts for 57–59% of each guided denoising step. The standard approach uses PyTorch autograd through dense transition matrix operations, storing $O(L \times |Q|^2)$ intermediate values on the autograd tape. This motivates exploring custom gradient implementations that could reduce memory traffic and enable kernel fusion.

We hypothesize that custom forward-backward vector-Jacobian products (VJPs) can accelerate DFA-guided diffusion by analytically computing gradients without autograd tape storage. If tokenizer-aligned DFAs are sparse, edge-list representations could further reduce computation. We implement this approach using Triton kernels and pre-allocated buffers, targeting $\geq 3\times$ end-to-end speedup over `torch.compile` baselines.

Our experiments reveal a negative result: the custom VJP achieves only 1.01–1.23× speedup, far below our target. The root cause is that tokenizer-aligned DFAs are inherently dense, with 50–6,177 edges per state-pair, invalidating the sparse optimization hypothesis. Our contributions are:

[1] https://gitlab.com/fars-a/fb-diffinity-sparse-grad

- We implement custom forward-backward VJPs for DFA acceptance probability with Triton kernels and adaptive dispatch, producing numerically identical gradients to autograd (cosine similarity 1.0, relative L2 error 1.7×10^{-5}).
- We evaluate on 4 constraint types $\times$ 2 batch sizes, finding limited speedups (1.01–1.23$\times$) that fall short of our 3$\times$ target.
- We identify the root cause: tokenizer-aligned DFAs have 65K–348K edges despite only 6–68 states, making transition matrices effectively fully dense.
- We document this negative result to inform future optimization efforts, suggesting that alternative approaches such as state-space reduction or approximate inference may be more promising.

2 Related Work

Diffusion Language Models. Diffusion models have emerged as a powerful paradigm for generative modeling, with foundational work establishing denoising diffusion probabilistic models (Ho et al., 2020) and score-based generative modeling through stochastic differential equations (Song et al., 2020). Extending these methods to discrete text domains has proven challenging. Early approaches such as Diffusion-LM (Li et al., 2022) operate in continuous embedding space, enabling gradient-based controllable generation. Discrete diffusion models (Austin et al., 2021) directly model token sequences through structured state-space transitions. More recent work has achieved competitive likelihoods with autoregressive models: SEDD (Lou et al., 2024) introduces score entropy for discrete diffusion, while PLAID (Gulrajani & Hashimoto, 2023) demonstrates that diffusion language models can outperform GPT-2 on standard benchmarks through algorithmic improvements and scaling.

Constrained Text Generation. Controlling neural text generation to satisfy structural constraints remains an active research area. For autoregressive models, NeuroLogic Decoding (Lu et al., 2021) enables lexical constraints through beam search modifications, while COLD decoding (Qin et al., 2022) formulates constraints as energy functions optimized via Langevin dynamics. Grammar-constrained decoding approaches (Ugare et al., 2024; Park et al., 2025; Sun et al., 2025) ensure outputs conform to context-free grammars. For diffusion models, recent work has explored DFA and CFG guidance: Mündler et al. (2025) propose constrained decoding for diffusion LLMs with context-free grammars, DINGO (Suresh et al., 2025) introduces constrained inference through DFA guidance, and Kim et al. (2026) demonstrate that continuous diffusion models can obey formal syntax. These methods compute DFA acceptance probabilities during denoising, which becomes a computational bottleneck that motivates our optimization efforts.

Differentiable Finite-State Methods. The forward-backward algorithm for computing gradients through finite-state computations has a rich history in structured prediction. Eisner (2016) establishes the connection between inside-outside algorithms and backpropagation. Semiring parsing (Goodman, 1999) provides a unified framework for dynamic programming over weighted grammars. Modern deep learning libraries have incorporated differentiable finite-state operations: Torch-Struct (Rush, 2020) provides GPU-accelerated structured prediction primitives, while differentiable weighted finite-state transducers (Hannun et al., 2020) enable end-to-end training of speech recognition systems. LAST (Wu et al., 2023) demonstrates scalable lattice-based speech modeling in JAX. Our work builds on these foundations by implementing custom VJPs for DFA acceptance probability computation, though we find that the dense structure of tokenizer-aligned DFAs limits the achievable speedups.

3 Method

We describe our custom forward-backward VJP for computing gradients of DFA acceptance probability in continuous diffusion language models. Our approach replaces PyTorch's autograd tape with analytic gradient computation, aiming to reduce memory traffic and enable kernel fusion.

3.1 Background: DFA Acceptance Probability

Given a tokenizer-aligned DFA $A = (\Sigma, Q, q_0, \delta, F)$ where Σ is the token vocabulary, Q is the state set, q_0 is the initial state, δ is the transition function, and $F \subseteq Q$ is the set of accepting states. For a length-L sequence with per-position token distributions $p_k(\text{tok})$ produced by the diffusion model decoder, the expected acceptance probability is:

$$Z = \mathbb{E}_{s\sim\prod_k p_k}\left[\mathbf{1}[s \in L(A)]\right] \tag{1}$$

This is computed via the forward algorithm in the probability semiring. Define the transition matrix $M_k \in \mathbb{R}^{|Q|\times|Q|}$ at position k where $M_k[i,j] = \sum_{\text{tok}:\delta(i,\text{tok})=j} p_k(\text{tok})$. The forward messages $\alpha_k \in \mathbb{R}^{|Q|}$ are computed recursively:

$$\alpha_0 = \mathbf{e}_{q_0}, \quad \alpha_k = M_k^\top \alpha_{k-1} \tag{2}$$

where $\mathbf{e}_{q_0}$ is the one-hot vector for the initial state. The acceptance probability is $Z = \sum_{q\in F} \alpha_L(q)$.

3.2 Baseline Autograd Implementation

The standard PyTorch implementation constructs M_k matrices at each position and uses automatic differentiation to compute gradients. This approach stores $O(L \times |Q|^2)$ intermediate values on the autograd tape, creating memory overhead and limiting kernel fusion opportunities. For tokenizer-aligned DFAs with large edge sets, this becomes a significant bottleneck.

3.3 Custom Forward-Backward VJP

Our custom `autograd.Function` computes gradients analytically using the forward-backward algorithm, eliminating autograd tape storage. The backward messages $\beta_k \in \mathbb{R}^{|Q|}$ are computed in reverse:

$$\beta_L(q) = \mathbf{1}[q \in F], \quad \beta_{k-1} = M_k \beta_k \tag{3}$$

For each position k and token tok, the gradient of Z with respect to the token probability is:

$$\frac{\partial Z}{\partial p_k(\text{tok})} = \sum_{(q \xrightarrow{\text{tok}} q')\in E} \alpha_{k-1}(q)\cdot \beta_k(q') \tag{4}$$

where E is the set of DFA transitions. This arc posterior formulation allows computing gradients by iterating over edges rather than storing intermediate matrices. Figure 1 illustrates the comparison between the baseline autograd approach and our custom forward-backward VJP.

3.4 Implementation Details

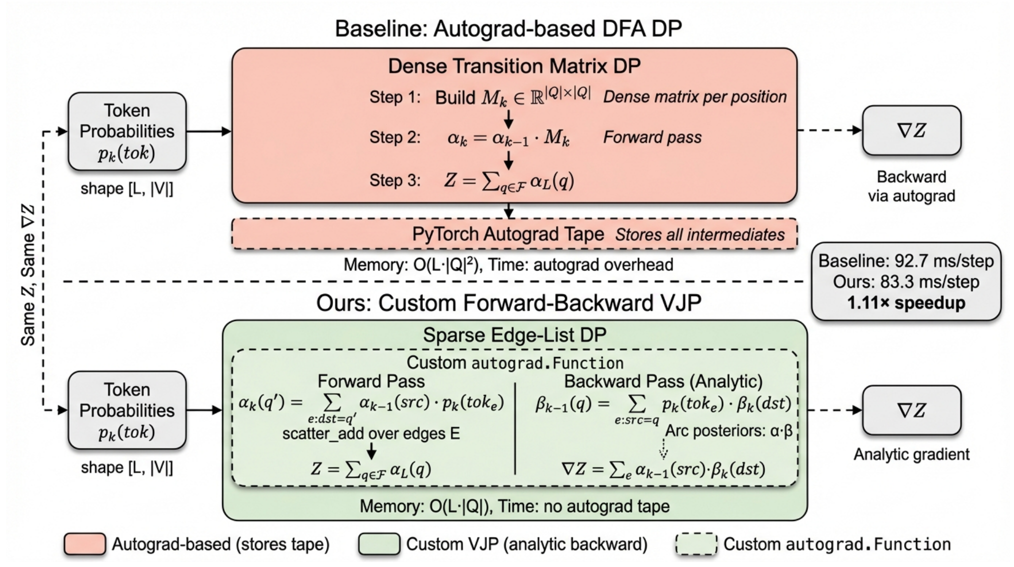
Our implementation includes several optimizations. For small-$|Q|$ DFAs ($|Q|^2 \leq 200$), we use custom Triton kernels for transition matrix construction that fuse the gather-scatter operations. For larger DFAs, we use PyTorch's native scatter-add operations with pre-allocated buffers to avoid repeated memory allocation. The implementation adaptively dispatches between these strategies based on DFA size.

We maintain numerical stability by operating in log-space for the forward pass and converting to probability space only for the final gradient computation. The backward pass reuses the forward messages stored during the forward pass, requiring only $O(L\times|Q|)$ additional memory for backward messages rather than $O(L \times |Q|^2)$ for full transition matrices.

4 Experiments

4.1 Experimental Setup

We evaluate our custom VJP implementation on PLAID 1.3B (Gulrajani & Hashimoto, 2023), a continuous diffusion language model with 32,768 vocabulary size and 64-token sequence length. We use DIFFINITY-style guidance (Kim et al., 2026) with $\gamma = 2.5$ and score temperature 0.9.



Baseline: Autograd-based DFA DP
Token Probabilities $p_k(tok)$
shape [L, |V|]
Dense Transition Matrix DP
Step 1: Build $M_k \in \mathbb{R}^{|Q|\times|Q|}$ Dense matrix per position
Step 2: $\alpha_k = \alpha_{k-1} \cdot M_k$ Forward pass
Step 3: $Z = \sum_{q\in\mathcal{F}} \alpha_L(q)$
PyTorch Autograd Tape Stores all intermediates
Memory: O(L·|Q|²), Time: autograd overhead
∇Z
Backward via autograd
Baseline: 92.7 ms/step
Ours: 83.3 ms/step
1.11× speedup
Same Z, Same ∇Z
Ours: Custom Forward-Backward VJP
Sparse Edge-List DP
Custom autograd.Function
Forward Pass
$\alpha_k(q') = \sum_{e:dst=q'} \alpha_{k-1}(src)\cdot p_k(tok_e)$
scatter_add over edges E
$Z = \sum_{q\in\mathcal{F}} \alpha_L(q)$
Backward Pass (Analytic)
$\beta_{k-1}(q) = \sum_{e:src=q} p_k(tok_e)\cdot\beta_k(dst)$
Arc posteriors: α·β
$\nabla Z = \sum_e \alpha_{k-1}(src)\cdot\beta_k(dst)$
Memory: O(L·|Q|), Time: no autograd tape
Analytic gradient
Autograd-based (stores tape)
Custom VJP (analytic backward)
Custom autograd.Function

Figure 1: Comparison of baseline autograd (left) and custom forward-backward VJP (right) for computing gradients of DFA acceptance probability. The baseline relies on PyTorch's autograd tape to store intermediate matrices, while the custom VJP analytically derives gradients using the forward-backward algorithm, eliminating tape storage.

Table 1: End-to-end step time (ms) and speedup comparison across methods, constraints, and batch sizes. Best speedup per column in **bold**. Custom VJP achieves only 1.01–1.23× step speedup, far below the 3× target.

	Batch Size 1				Batch Size 4			
Method	json_s	json_l	nl_pre	nl_btw	json_s	json_l	nl_pre	nl_btw
Baseline A	92.7	95.8	100.7	97.9	93.6	99.4	114.7	127.8
Baseline B	90.7	91.9	94.1	95.8	90.8	97.7	116.0	119.9
Custom VJP	83.3	83.4	93.4	97.0	87.7	86.5	96.8	97.9
Speedup (vs B)	**1.09**×	**1.10**×	1.01×	0.99×	1.04×	**1.13**×	**1.20**×	**1.23**×

We evaluate on four pre-registered constraint types spanning different DFA complexities: two JSON schema constraints (`json_smallest` with 20 states and `json_largest` with 68 states) and two natural language regex patterns (`nl_prefix` with 6 states and `nl_between` with 10 states). JSON constraints use 256 denoising timesteps while NL regex constraints use 1024 timesteps. We test batch sizes of 1 and 4, with timing averaged over 3 random seeds and satisfaction rates computed from 50 samples per constraint.

We compare three methods: **Baseline A** (original autograd DP), **Baseline B** (`torch.compile` with inductor backend, selected as the best-performing simple optimization), and **Custom VJP** (our analytic forward-backward implementation). Our pre-registered targets were $\geq 3\times$ end-to-end step speedup and $\geq 10\times$ DFA subroutine speedup over Baseline B.

4.2 Main Results

Table 1 presents the end-to-end step times and speedups across all configurations. The custom VJP achieves only 1.01–1.23× speedup over the `torch.compile` baseline, far below our 3× target. The best speedup (1.23×) occurs on `nl_between` at batch size 4, while `nl_between` at batch size 1 shows a slight slowdown (0.99×), indicating that overhead can exceed gains in some configurations.

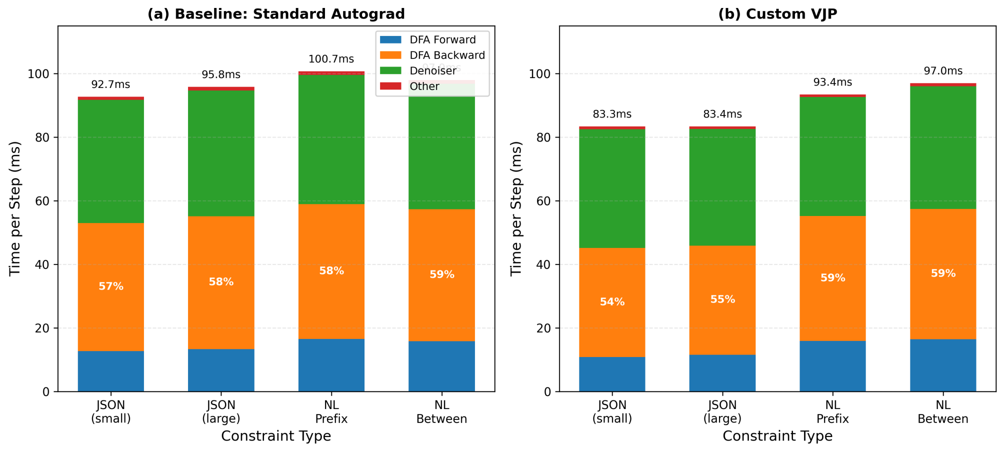


Figure 2: Time breakdown per guided denoising step comparing baseline autograd (left) and custom VJP (right) across four constraint types. DFA computation (forward + backward) accounts for 54–59% of step time. The custom VJP achieves modest speedups (1.01–1.15×) at batch size 1.

Table 2: DFA statistics for each constraint type. Tokenizer-aligned DFAs are inherently dense, with 50–6,177 edges per state-pair, invalidating the sparse edge-list hypothesis.

Constraint	States ($\|Q\|$)	Edges ($\|E\|$)	Density ($\|E\|/\|Q\|^2$)	Description
json_smallest	20	65,590	164.0	`{"ok": true|false}`
json_largest	68	235,114	50.9	5-field string schema
nl_prefix	6	222,363	**6,177**	`[A-Za-z ]*said...`
nl_between	10	347,811	3,478	`...know...will...`

4.3 Profiling Analysis

Figure 2 shows the time breakdown per guided denoising step. DFA computation (forward + backward) accounts for 57–59% of step time in the baseline, confirming that the DFA subroutine is indeed the primary bottleneck rather than the PLAID denoiser. However, the custom VJP achieves only modest reductions in DFA time, with the overall step time remaining similar between methods.

4.4 Root Cause: Dense DFA Structure

Table 2 reveals the root cause of limited speedups: tokenizer-aligned DFAs are inherently dense. Despite having only 6–68 states, these DFAs contain 65,590–347,811 edges, yielding densities of 50–6,177 edges per state-pair. This invalidates our sparse edge-list hypothesis—the transition matrices are effectively fully dense, leaving little room for optimization beyond what `torch.compile` already achieves.

The high density arises from tokenizer alignment: when a character-level DFA is converted to accept token sequences, each state-pair must account for all tokens whose character sequences are valid transitions. Natural language patterns with broad character classes (e.g., `[A-Za-z ]*`) result in nearly every token being a valid transition from most states.

4.5 Gradient Correctness and Satisfaction

The custom VJP produces numerically identical gradients to the baseline autograd implementation, with cosine similarity of 1.0 and relative L2 error of 1.7×10^{-5}, validating our implementation correctness.

Table 3: Constraint satisfaction rates (%) comparing baseline and custom VJP. Rates are preserved within sampling noise ($n = 50$ samples, 95% CI width ≈8pp).

Constraint	Baseline A	Custom VJP	Δ
json_smallest	98%	90%	−8pp
json_largest	0%	0%	0pp
nl_prefix	88%	88%	0pp
nl_between	78%	78%	0pp

Table 3 shows that constraint satisfaction rates are preserved within sampling noise. Three of four constraints show identical rates (0pp change), while `json_smallest` shows an 8pp variation consistent with the expected 95% confidence interval width for $n = 50$ samples.

5 Conclusion

We implemented custom forward-backward VJPs for DFA acceptance probability computation in continuous diffusion language models, achieving only 1.01–1.23× speedup over `torch.compile`—far below our 3× target. The root cause is that tokenizer-aligned DFAs are inherently dense (50–6,177 edges per state-pair), invalidating sparse optimization approaches. This negative result suggests that future work on accelerating DFA-guided diffusion should explore alternative directions such as state-space reduction, approximate inference, or learned constraint representations rather than gradient computation optimizations.

References

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. *ArXiv*, abs/2107.03006, 2021.

Jason Eisner. Inside-outside and forward-backward algorithms are just backprop (tutorial paper). pp. 1–17, 2016.

Joshua Goodman. Semiring parsing. *Comput. Linguistics*, 25:573–605, 1999.

Ishaan Gulrajani and Tatsunori Hashimoto. Likelihood-based diffusion language models. *ArXiv*, abs/2305.18619, 2023.

Awni Y. Hannun, Vineel Pratap, Jacob Kahn, and Wei-Ning Hsu. Differentiable weighted finite-state transducers. *ArXiv*, abs/2010.01003, 2020.

Jonathan Ho, Ajay Jain, and P. Abbeel. Denoising diffusion probabilistic models. *ArXiv*, abs/2006.11239, 2020.

Jinwoo Kim, Taylor Berg-Kirkpatrick, and Loris D'antoni. Continuous diffusion models can obey formal syntax. 2026.

Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy Liang, and Tatsunori B. Hashimoto. Diffusion-lm improves controllable text generation. In *Advances in Neural Information Processing Systems*, 2022.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. In *International Conference on Machine Learning*, 2024.

Ximing Lu, Peter West, Rowan Zellers, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Neurologic decoding: (un)supervised neural text generation with predicate logic constraints. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021.

Niels Mündler, Jasper Dekoninck, and Martin Vechev. Constrained decoding of diffusion llms with context-free grammars, 2025.

Kanghee Park, Timothy Zhou, and Loris D'antoni. Flexible and efficient grammar-constrained decoding. *ArXiv*, abs/2502.05111, 2025.

Lianhui Qin, Sean Welleck, Daniel Khashabi, and Yejin Choi. Cold decoding: Energy-based constrained text generation with langevin dynamics. In *Advances in Neural Information Processing Systems*, 2022.

Alexander M. Rush. Torch-struct: Deep structured prediction library. *ArXiv*, abs/2002.00876, 2020.

Yang Song, Jascha Narain Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. *ArXiv*, abs/2011.13456, 2020.

Xintong Sun, Chi Wei, Minghao Tian, and Shiwen Ni. Earley-driven dynamic pruning for efficient structured decoding. *ArXiv*, abs/2506.01151, 2025.

Tarun Suresh, Debangshu Banerjee, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. Dingo: Constrained inference for diffusion llms, 2025.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. Grammar-aligned decoding. In *International Conference on Machine Learning*, 2024.

Ke Wu, Ehsan Variani, Tom Bagby, and M. Riley. Last: Scalable lattice-based speech modelling in jax. *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1–5, 2023.