

The Thesis Committee for Nathan Barry
certifies that this is the approved version of the following thesis:

Block-Wise KV Caching for Uniform Diffusion Language Models

SUPERVISING COMMITTEE:

Aditya Akella, Supervisor

Neeraja J. Yadwadkar

**Block-Wise KV Caching for Uniform Diffusion Language
Models**

by
Nathan Barry

Thesis

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

**The University of Texas at Austin
May 2026**

Abstract

Block-Wise KV Caching for Uniform Diffusion Language Models

Nathan Barry, MS
The University of Texas at Austin, 2026

SUPERVISOR: Aditya Akella

Diffusion language models offer a parallel alternative to autoregressive generation: rather than predicting tokens left-to-right, they iteratively refine an entire sequence from noise in a fixed number of steps. (Austin et al., 2021; Hoogeboom et al., 2021). Among these, uniform diffusion models have shown promise for their self-correction capability, few-step generation ability, and better scaling properties compared to masked diffusion models (von Rütte et al., 2025a,b). However, the iterative denoising process requires a full transformer forward pass at every step, creating a major computational bottleneck. Prior work has applied KV caching for masked diffusion models (Wu et al., 2025; Ma et al., 2025; Liu et al., 2025; Nguyen-Tri et al., 2025), but it has remained an open question whether similar approaches would work for uniform diffusion models.

We present **Uniform-Cache**, a simple and effective KV caching mechanism for uniform diffusion language models based on block-wise generation. On the first step of each block, we recompute the entire sequence to update KV states which are reused for every subsequent step within the block. After the first full forward pass that populates the cache, each subsequent step recomputes only the current block’s tokens, attending them against the full cached sequence. We show empirically that

KV drift is semi-local: it is concentrated in positions near recent token changes, making it highest in the current block and decaying sharply for both past and future blocks. Completed blocks receive 24–1000× lower drift than the current block while accounting for ~26% of attention mass; the prompt and future blocks are similarly stable.

Evaluated on the GIDD 3B and 10B uniform diffusion models, Uniform-Cache achieves up to **12.2× speedup** over the uncached baseline with negligible impact on measured generative perplexity. Operation-level profiling shows attention computation reduces by 13.4×. Our empirical analysis of token dynamics, KV drift patterns, entropy evolution, and the interaction between sampling strategies and caching provides grounded empirical support for each design decision and offers broader insight into the mechanics of uniform diffusion denoising.

Table of Contents

List of Tables	8
List of Figures	9
Chapter 1: Introduction	10
1.1 Motivation	10
1.2 Problem Statement	11
1.3 Contributions	12
1.4 Thesis Organization	12
Chapter 2: Background	13
2.1 Discrete Diffusion Models	13
2.1.1 Forward Process	13
2.1.2 Reverse Process	14
2.1.3 Training Objective	15
2.1.4 Transition Matrices	15
2.2 Noise Types in Discrete Diffusion	15
2.2.1 Masked (Absorbing-State) Diffusion	16
2.2.2 Uniform Diffusion	17
2.2.3 GIDD: Generalized Interpolating Discrete Diffusion	18
2.2.4 Scaling Properties	19
2.3 Block-Wise Denoising	19
2.3.1 GIDD’s Attention Mask	19
Chapter 3: Related Work	21
3.1 Diffusion Language Model Foundations	21
3.2 KV Caching for Diffusion Models	22
Chapter 4: Method	25
4.1 Problem Formulation	25
4.2 Three Generation Strategies	25
4.2.1 Baseline (No Cache)	26
4.2.2 Prefix Cache	26
4.2.3 Uniform Cache	27
4.3 Algorithm	28
4.4 Design Decisions	29
4.4.1 Block-Only Recompute	29

4.4.2	Full Forward on First Step of Each Block	30
4.4.3	Recompute Interval for the Next Block	30
4.4.4	Adaptive Sampling as Default	31
Chapter 5:	Experimental Setup	33
5.1	Models	33
5.2	Hardware	33
5.3	Datasets	33
5.4	Evaluation Metrics	33
5.4.1	Quality: Generative Perplexity	33
5.4.2	Efficiency: Wall-Clock Time	34
5.4.3	Throughput	34
5.4.4	Operation-Level Profiling	34
5.5	Hyperparameters	34
5.6	Experiments	35
Chapter 6:	Results	37
6.1	Main Benchmark	37
6.2	Operation-Level Profiling	39
6.3	Batching Throughput	41
6.4	Step Degradation	41
6.5	Analysis of Token Dynamics	42
6.5.1	Token Change Rates	42
6.5.2	Entropy Convergence	43
6.6	KV Drift Analysis	44
6.6.1	Key–Value Drift Correlation	44
6.6.2	Drift by Block Region	45
6.6.3	Attention Mass by Region	46
6.6.4	Cross-Layer Drift Correlation	47
6.7	Ablations	48
6.7.1	Block Size	48
6.7.2	Recompute Interval	48
6.7.3	Tokens per Step (n_{tokens})	49
6.8	Generalization to 10B Model	49
Chapter 7:	Conclusion	50
7.1	Summary of Contributions	50
7.2	Limitations	51
7.3	Future Work	52

Appendix A: Intra-Block Denoising Dynamics	54
A.1 Entropy Convergence Within Blocks (s32/b32)	54
A.2 KV Drift by Denoising Timestep (s32/b32)	55
Appendix B: Ancestral Sampling with Block-Wise Generation	56
Appendix C: Sample Generation Outputs	57
Appendix D: bfloat16 Numerical Artifact: Baseline vs. Prefix Cache PPL Gap	59
Works Cited	60

List of Tables

3.1	KV caching methods for masked diffusion language models	23
5.1	Default hyperparameters for all experiments.	35
6.1	End-to-end benchmark results	37
6.2	Per-operation timing for s32/b32 adaptive	39
6.3	Entropy convergence at s512/b512	43
6.4	Key vs. value drift	44
6.5	Key-value drift correlation	44
6.6	KV drift by block region	45
6.7	Attention mass by region	47
6.8	Cross-layer drift correlation	47
6.9	Block size ablation	48
6.10	Recompute interval ablation	48
6.11	n_{tokens} ablation	49
6.12	10B model benchmark results	49
B.1	Ancestral sampling benchmark results	56

List of Figures

4.1	Sequence regions recomputed vs. cached per step for each generation strategy	26
4.2	Token change rate: ancestral sampling	31
6.1	Benchmark results: generation time and perplexity at s32/b32	38
6.2	Per-operation CUDA timing breakdown at s32/b32	40
6.3	Per-step operation cost over the course of generation at s32/b32	40
6.4	Batching throughput vs. batch size at s32/b32	41
6.5	Quality vs. step count under ancestral sampling	42
6.6	Entropy convergence at s512/b512: adaptive (left) and ancestral (right)	43
6.7	KV drift and attention mass by block region at s32/b32	46
A.1	Entropy and KV drift diagnostics at s32/b32	54
A.2	KV drift by denoising timestep at s32/b32	55

Chapter 1: Introduction

1.1 Motivation

Language modeling has been dominated by autoregressive (AR) architectures (Radford et al., 2019; Vaswani et al., 2017). While remarkably effective, this sequential generation paradigm, generating text one token at a time from left to right, has inherent limitations: it cannot revise earlier tokens in light of later context, and generation latency scales linearly with sequence length.

Diffusion language models offer a fundamentally different approach (Austin et al., 2021; Hoogeboom et al., 2021). Rather than committing to tokens sequentially, these models begin with a fully noised sequence and iteratively denoise all positions in parallel over many steps. Among discrete diffusion models, two noise processes have emerged as dominant. *Masked diffusion* replaces tokens with a dedicated [MASK] token, progressively unmasking them during generation (Austin et al., 2021; Sahoo et al., 2024). *Uniform diffusion* replaces tokens with uniformly random vocabulary items, which forces the model to identify and correct errors rather than simply fill in blanks (Austin et al., 2021; Hoogeboom et al., 2021).

In masked diffusion, each token can only transition from masked to unmasked, so once decoded it is permanently fixed. In uniform diffusion, tokens can transition between any two vocabulary items at any denoising step, allowing the model to revise earlier decisions throughout generation. Recent scaling studies find that the likelihood gap between uniform and masked diffusion narrows substantially as compute scales, suggesting uniform diffusion becomes increasingly competitive at large model sizes (von Rütte et al., 2025b).

The iterative nature of diffusion generation creates a major computational bottleneck. Each denoising step requires a full forward pass through the entire transformer for all positions. This computational cost is the primary barrier to practical

deployment of diffusion language models.

1.2 Problem Statement

In autoregressive models, KV caching is the standard technique for avoiding redundant computation: once a token’s key and value projections are computed, they are cached and reused for all subsequent tokens. This works because the causal attention mask guarantees that earlier positions’ representations never change once computed.

Diffusion language models use bidirectional attention, so all positions affect one another at every step. This appears to make KV caching impossible in principle: if all positions are interdependent, every position’s representation is subject to change at every step.

Recent work has shown that reusing stale KV projections across denoising steps is possible without meaningfully degrading generation quality, yielding large speedups. The central question then becomes: which positions should be cached, and when should they be refreshed? For masked diffusion, the absorbing state provides a natural answer: once a token transitions from [MASK] to a decoded state irreversibly, its KV projections stabilize, making decoded positions obvious candidates for caching. Building on this, Fast-dLLM (Wu et al., 2025) achieves up to $27.6\times$ speedup on LLaDA, dKV-Cache (Ma et al., 2025) achieves $1.8\text{--}2.4\times$, dLLM-Cache (Liu et al., 2025) achieves up to $9.1\times$, and Elastic-Cache (Nguyen-Tri et al., 2025) claims up to $45\times$.

Uniform diffusion offers no such structural guidance. Any token can be revised at any step — there is no decode event, no absorbing state, no structural indicator that any position may be stable. Prior acceleration work on uniform diffusion has therefore focused on reducing the number of denoising steps rather than caching. Whether KV caching opportunities existed for uniform diffusion has remained an open question.

1.3 Contributions

This thesis makes the following contributions:

1. **An empirical analysis of uniform diffusion denoising dynamics.** We characterize KV drift, attention mass, entropy convergence, and token change rates throughout block-wise generation, showing that drift is semi-local and sharply concentrated in the current block, and that adaptive sampling produces lower-entropy, more cache-friendly distributions than ancestral sampling. Together these observations provide a grounded empirical justification for block-only caching in the absence of any structural guarantee.
2. **Uniform-Cache, the first inference-time KV caching method for uniform diffusion language models.** We introduce block-wise caching that runs a full forward pass on the first step of each block, then recomputes only the current block for all subsequent steps, demonstrating that KV caching is viable for uniform diffusion despite the absence of an absorbing state. With adaptive sampling, the method achieves up to $12.2\times$ speedup while consistently reducing measured generative perplexity across all tested configurations, and generalizes consistently to the 10B GIDD model.

1.4 Thesis Organization

Chapter 2 provides background on discrete diffusion models, contrasting masked and uniform noise processes. Chapter 3 surveys related work on diffusion language model foundations, scaling behavior, and existing KV caching methods. Chapter 4 presents our method, including the baseline, prefix cache, and Uniform-Cache generation strategies, the block-only recompute algorithm, and the design decisions justified by our empirical analysis. Chapter 5 describes our experimental setup. Chapter 6 presents results across all experiments. Chapter 7 concludes with a summary of contributions, limitations, and directions for future work.

Chapter 2: Background

2.1 Discrete Diffusion Models

Diffusion models generate data by learning to reverse a gradual noising process. In the continuous domain, this approach has achieved remarkable success in image generation (Ho et al., 2020; Song et al., 2021). Adapting diffusion to discrete categorical data (such as text, where each position takes one of K unordered vocabulary tokens) requires replacing the continuous Gaussian noise process with discrete transition matrices over the vocabulary. While Austin et al. (2021) originally formulated this framework in discrete time, the resulting transition matrices can equivalently be viewed as discretizations of an underlying continuous-time Markov chain (Campbell et al., 2022; von Rütte et al., 2025a), a perspective we adopt in later chapters.

2.1.1 Forward Process

The forward (noising) process defines a Markov chain that progressively corrupts clean data \mathbf{x}_0 into increasingly noisy versions $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T$. At each timestep t , each token position independently transitions according to a categorical distribution parameterized by a transition matrix $\mathbf{Q}_t \in \mathbb{R}^{K \times K}$:

$$q(\mathbf{x}_t | \mathbf{x}_{t-1}) = \text{Cat}(\mathbf{x}_t; \mathbf{x}_{t-1} \mathbf{Q}_t) \quad (2.1)$$

where \mathbf{x}_{t-1} is a one-hot row vector and \mathbf{Q}_t is a row-stochastic matrix whose entry $[\mathbf{Q}_t]_{ij}$ gives the probability of transitioning from token i to token j . By the Markov property, the cumulative transition from \mathbf{x}_0 to \mathbf{x}_t is given by:

$$q(\mathbf{x}_t | \mathbf{x}_0) = \text{Cat}(\mathbf{x}_t; \mathbf{x}_0 \overline{\mathbf{Q}}_t), \quad \overline{\mathbf{Q}}_t = \mathbf{Q}_1 \mathbf{Q}_2 \cdots \mathbf{Q}_t \quad (2.2)$$

The noise schedule is chosen such that each row of $\overline{\mathbf{Q}}_T$ converges to the stationary distribution π , so that $q(\mathbf{x}_T | \mathbf{x}_0) = \text{Cat}(\mathbf{x}_T; \mathbf{x}_0 \overline{\mathbf{Q}}_T) \approx \pi$ independent of \mathbf{x}_0 .

2.1.2 Reverse Process

The reverse (denoising) process learns to undo the corruption. Rather than predicting the reverse transition $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ directly, it is standard practice (Ho et al., 2020; Hoogeboom et al., 2021; Austin et al., 2021) to use an x_0 -parameterization: a neural network \hat{p}_θ is trained to predict the original clean data from the noised input,

$$\hat{p}_\theta(\tilde{\mathbf{x}}_0|\mathbf{x}_t) \tag{2.3}$$

and the reverse transition is obtained by marginalizing over this prediction through the tractable posterior:

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \sum_{\tilde{\mathbf{x}}_0} q(\mathbf{x}_{t-1}|\mathbf{x}_t, \tilde{\mathbf{x}}_0) \cdot \hat{p}_\theta(\tilde{\mathbf{x}}_0|\mathbf{x}_t) \tag{2.4}$$

The posterior $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$ is obtained in closed form via Bayes' rule:

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \text{Cat}\left(\mathbf{x}_{t-1}; \frac{\mathbf{x}_t \mathbf{Q}_t^\top \odot \mathbf{x}_0 \overline{\mathbf{Q}}_{t-1}}{\mathbf{x}_0 \overline{\mathbf{Q}}_t \mathbf{x}_t^\top}\right) \tag{2.5}$$

This parameterization has two practical advantages over predicting $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ directly. First, it automatically inherits the sparsity pattern of \mathbf{Q}_t , so the learned reverse distribution only assigns mass to transitions that are feasible under the forward process. Second, it enables k -step inference by replacing $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \tilde{\mathbf{x}}_0)$ with $q(\mathbf{x}_{t-k}|\mathbf{x}_t, \tilde{\mathbf{x}}_0)$, allowing a trained model to trade off sample quality for inference speed at test time. Generation proceeds by sampling \mathbf{x}_T from the stationary distribution π and iteratively applying the learned reverse transitions.

2.1.3 Training Objective

The model is trained by maximizing a variational lower bound (ELBO) on the log-likelihood. The negative ELBO decomposes as:

$$\begin{aligned}
 L_{\text{vb}} = & \underbrace{D_{\text{KL}}(q(\mathbf{x}_T|\mathbf{x}_0) \parallel p(\mathbf{x}_T))}_{L_T} \\
 & + \sum_{t=2}^T \underbrace{\mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)}[D_{\text{KL}}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) \parallel p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))]}_{L_{t-1}} \\
 & - \underbrace{\mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}[\log p_\theta(\mathbf{x}_0|\mathbf{x}_1)]}_{L_0}
 \end{aligned} \tag{2.6}$$

The L_T term measures how close the fully noised distribution is to the prior and is approximately zero when the forward process has converged to its stationary distribution, so in practice only the L_{t-1} and L_0 terms contribute meaningfully to the gradient. Each L_{t-1} term compares the model’s reverse transition $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ against the closed-form posterior $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$, which under the \mathbf{x}_0 -parameterization is minimized exactly when \hat{p}_θ places all of its mass on the true \mathbf{x}_0 .

2.1.4 Transition Matrices

The choice of transition matrix \mathbf{Q}_t is the key design decision in discrete diffusion: it determines the stationary distribution of the chain, the structure of the tractable posterior, and the behavioral properties of the model during denoising. Austin et al. (2021) explored several families, including absorbing (masked), uniform, Gaussian, and embedding-based transitions. The two that have shown the most empirical promise for language modeling — absorbing and uniform — are treated in detail in Section 2.2.

2.2 Noise Types in Discrete Diffusion

The distinction between masked and uniform diffusion is central to this thesis, as it determines the nature of the caching opportunity. Throughout this section, x_t

and x_0 denote the token at a single position; the sequence-level distributions follow by independence across positions.

2.2.1 Masked (Absorbing-State) Diffusion

In masked diffusion, \mathbf{Q}_t maps each token to a special [MASK] token with some probability, and [MASK] is absorbing — it never transitions to any other token. Let \mathbf{m} denote the one-hot vector for the [MASK] token. The per-position noise distribution is:

$$q(x_t|x_0) = \text{Cat}(x_t; (1 - \beta_t)x_0 + \beta_t \mathbf{m}) \quad (2.7)$$

where β_t increases from 0 to 1, and the stationary distribution is \mathbf{m} . This creates a monotonic process: once a position is masked it stays masked, and during denoising the sequence is always partitioned into decoded positions (unmasked and fixed) and masked positions (actively being denoised).

Masked diffusion’s monotonicity made it tractable and interpretable early on, with a clean connection to BERT-style masked language modeling and a simple absorbing-state ELBO. MDLM (Sahoo et al., 2024) demonstrated that this framework achieves strong language modeling results, and LLaDA (Nie et al., 2025) and Dream (Ye et al., 2025) have since scaled it to 8B and 7B parameters respectively. More recently, commercial models have validated masked diffusion at scale: Inception Labs’ Mercury family (Inception Labs, 2025) achieves over 1000 tokens/second on commodity hardware, and Google DeepMind’s Gemini Diffusion (Google DeepMind, 2025) demonstrates benchmark performance comparable to much larger autoregressive models. This momentum has left uniform diffusion comparatively under-explored, though recent work on favorable scaling properties (von Rütte et al., 2025b) and self-correction has renewed interest. Whether uniform diffusion can benefit from inference-time optimizations such as KV caching remains an open question that this thesis addresses.

2.2.2 Uniform Diffusion

In uniform diffusion, \mathbf{Q}_t replaces each token with a uniformly random vocabulary item rather than a dedicated mask token. Let $\mathbf{u} = \frac{1}{K}\mathbf{1}$ denote the uniform distribution over the vocabulary. The per-position noise distribution is:

$$q(x_t|x_0) = \text{Cat}(x_t; (1 - \beta_t)x_0 + \beta_t \mathbf{u}) \quad (2.8)$$

where β_t is the noise level at time t and K is the vocabulary size. As t increases, $\beta_t \rightarrow 1$ and the distribution approaches \mathbf{u} .

This process has fundamentally different properties from masking:

1. **No absorbing state:** There is no special token that signals “this position is corrupted.” A noised token looks like any other vocabulary token. The model must infer which positions are corrupted from context alone.
2. **Self-correction:** Because tokens are replaced with random vocabulary items rather than a fixed mask, the model can revise any position at any denoising step. A token that appears correct at step t may be changed at step $t + 1$ if the model determines, given updated context, that a different token is more appropriate. Crucially, this means uniform diffusion does not suffer from the curse of parallel decoding (Wu et al., 2025) that affects masked diffusion.
3. **No convergence signal:** In masked diffusion, a block is unambiguously finished when all its positions have transitioned from [MASK] to content tokens. In uniform diffusion, no such signal exists: a position may hold the correct token at any step without any structural indication that it has stabilized. This makes it impossible to determine from \mathbf{x}_t alone whether a region has converged, which has direct implications for how generation must be structured.

The curse of parallel decoding (Wu et al., 2025). To accelerate masked diffusion, practitioners unmask multiple tokens simultaneously per step. However, this creates a problem: the joint probability of unmasking tokens at positions i and j is $p(x_i, x_j | \mathbf{x}_t) = p(x_i | \mathbf{x}_t) \cdot p(x_j | \mathbf{x}_t, x_i)$, but multi-token decoding samples them independently as $p(x_i | \mathbf{x}_t) \cdot p(x_j | \mathbf{x}_t)$, ignoring their correlation. The more tokens are unmasked simultaneously, the more this independence assumption degrades quality — imposing a fundamental limit on how aggressively the number of denoising steps can be reduced.

Uniform diffusion sidesteps this limit entirely. Because any token can be revised at any step, an error made when updating multiple tokens simultaneously is not permanent — the model can identify and correct it in a subsequent step. As a result, uniform diffusion is more robust to an aggressive reduction in the number of denoising steps than masked diffusion. Figure 6.5 in Section 6.4 confirms this empirically: under ancestral sampling, uniform diffusion degrades substantially more gracefully than masked diffusion as the step count decreases.

2.2.3 GIDD: Generalized Interpolating Discrete Diffusion

von Rütte et al. (2025a) introduce a generalized framework in which the forward marginal takes the form

$$q(x_t | x_0) = \text{Cat}(x_t; \alpha_t x_0 + \beta_t \pi_t) \tag{2.9}$$

where $\alpha_t = 1 - \beta_t$ is the signal coefficient, β_t is the noise level, and π_t is an arbitrary time-varying mixing distribution over the vocabulary. Setting $\pi_t = \mathbf{m}$ recovers masked diffusion exactly; setting $\pi_t = \mathbf{u}$ gives pure uniform diffusion. More generally, π_t can interpolate between these extremes, enabling hybrid noise processes. von Rütte et al. (2025a) derive closed-form cumulative transitions and a corresponding ELBO for any valid choice of α_t and π_t , yielding a unified training objective.

Their key empirical finding is that incorporating uniform noise unlocks *self-correction*: because every token must be evaluated for correctness during training,

the model learns to identify and revise its own errors at inference time. This capability is absent in masked-only training, where unmasked tokens are always assumed clean. Although von Rütte et al. (2025a) demonstrate this primarily with hybrid mask+uniform schedules, the underlying mechanism applies to pure uniform diffusion as well.

This thesis targets the pure uniform setting, using the `dvrulette/gidd-unif-3b` model released as part of the subsequent scaling study (von Rütte et al., 2025b), which instantiates the framework with $\pi_t = \mathbf{u}$.

2.2.4 Scaling Properties

von Rütte et al. (2025b) conducted the first comprehensive scaling study of discrete diffusion language models across noise types, training models from 200M to 10B parameters. They find that uniform diffusion requires more parameters but less data than masked diffusion to achieve comparable performance at compute-optimal scale. The likelihood gap between the two narrows from 3.2% at 10^{18} FLOPs to 1.7% at 10^{21} FLOPs, suggesting uniform diffusion becomes increasingly competitive as scale grows.

2.3 Block-Wise Denoising

Block-wise generation is a strategy for discrete diffusion models in which the response is divided into consecutive non-overlapping blocks of length B , each denoised independently over T steps (Nie et al., 2025). At each step, the model runs a full forward pass over the entire sequence and updates token predictions for the current block only.

2.3.1 GIDD’s Attention Mask

GIDD adds an asymmetric attention mask baked into training: current and future block positions attend to all positions, while prompt and completed block

positions attend only to each other. This enables partial KV reuse, but comes with significant limitations: it requires training modifications, forces a full forward pass at every block boundary, and prevents revision of completed tokens for previous blocks — sacrificing part of uniform diffusion’s self-correction capability.

This is also a fundamentally different approach from the KV caching methods that achieve large speedups in masked diffusion (Wu et al., 2025; Ma et al., 2025; Liu et al., 2025; Nguyen-Tri et al., 2025). Those are drop-in inference-time engines that exploit the empirical observation that KV values do not drift significantly between steps, reusing stale projections as an approximation. GIDD’s mask eliminates drift for a subset of positions by construction, but does not exploit step-to-step KV approximation — the mechanism behind the large speedups those methods achieve.

Chapter 3: Related Work

3.1 Diffusion Language Model Foundations

The development of effective discrete diffusion models for language has progressed rapidly. We survey the key developments that form the foundation for this work.

Multinomial Diffusion (Hooigeboom et al., 2021) was the first dedicated treatment of categorical diffusion with a uniform noise forward process, establishing the mathematical foundations that later work builds on.

D3PM (Austin et al., 2021) introduced the general framework for discrete diffusion with structured transition matrices, including absorbing, uniform, and discretized Gaussian transitions, and established the formal connection between masked language models and absorbing-state diffusion. Empirically, absorbing transitions outperformed uniform (1.45 vs. 1.61 bits/char on text8), which led subsequent work to focus on masked diffusion.

SEDD (Lou et al., 2024) introduced score entropy discrete diffusion, parameterizing the reverse process via concrete scores (probability ratios $p_t(y)/p_t(x)$), and was the first non-autoregressive model to match GPT-2 on zero-shot perplexity benchmarks.

MDLM (Sahoo et al., 2024) showed that the ELBO for absorbing-state diffusion reduces to a weighted average of masked language modeling losses, establishing a clean connection between BERT-style training and generative diffusion.

LLaDA (Nie et al., 2025) and **Dream** (Ye et al., 2025) scaled masked diffusion to 8B and 7B parameters respectively, demonstrating practical viability at scale and serving as the primary targets for the masked-diffusion caching methods discussed below.

GIDD (von Rütte et al., 2025a) extended the discrete diffusion framework to arbitrary interpolating noise processes. The key empirical contribution is quantifying self-correction under hybrid mask+uniform noise: generative perplexity improved by up to 55% and GPT-4o quality ratings improved across clarity, grammaticality, and factuality, while applying the same inference procedure to a mask-only model actively degraded quality.

Scaling Behavior (von Rütte et al., 2025b) provides the first comprehensive scaling study of discrete diffusion across noise types, from 25M to 10B parameters. The likelihood gap between uniform and masked diffusion narrows from 3.2% at 10^{18} FLOPs to 1.7% at 10^{21} FLOPs, and the 10B uniform model matches autoregressive scaling trends, suggesting the gap between uniform diffusion and autoregressive models continues to close at scale.

3.2 KV Caching for Diffusion Models

Several methods have developed KV caching for diffusion language models. All target masked diffusion models.

Fast-dLLM (Wu et al., 2025) introduces block-wise KV caching and confidence-aware parallel decoding for LLaDA. Since only the current block’s tokens change identity during decoding, the prompt and suffix KV activations are structurally stable and can be cached. The confidence-aware component dynamically selects how many tokens to decode per step based on model confidence. Together these achieve up to $27.6\times$ speedup on LLaDA. Fast-dLLM is the most closely related prior work: our caching design shares the block-wise structure, but extends it to uniform diffusion where tokens have no absorbing state, making the effectiveness of caching non-obvious a priori.

dKV-Cache (Ma et al., 2025) observes that once a token decodes from [MASK] its KV representation stabilizes, while masked tokens continue to fluctuate. This motivates caching decoded tokens and recomputing masked tokens at every

step. The key refinement is a one-step delay: the largest KV change occurs at the decode step itself, so caching is deferred one additional step to avoid the instability at the moment of transition. This achieves $1.83\text{--}2.42\times$ speedup on LLaDA-8B, and requires the discrete decode event that has no analogue in uniform diffusion.

dLLM-Cache (Liu et al., 2025) observes that prompt tokens are entirely static across denoising steps and experience low KV drift, so their features are cached and refreshed only every K_p steps. Second, most response tokens remain stable between adjacent steps; the V-verify mechanism identifies which ones have drifted by computing value vector cosine similarity at each layer, then recomputes only the top- ρ fraction with the most drift. This achieves up to $9.1\times$ speedup on LLaDA-8B.

Elastic-Cache (Nguyen-Tri et al., 2025) introduces an adaptive, layer-aware KV cache refresh policy. The core observations are that KV drift increases with transformer depth and that the most-attended token has minimal drift, making its attention weights a cheap proxy for cache staleness. At each step, Elastic-Cache monitors whether the most-attended token’s attention weights have changed significantly; if so, it recomputes only the deeper layers where drift is highest, reusing shallow-layer caches. This achieves up to $45.1\times$ speedup on LLaDA-1.5.

Method	Noise Type	Speedup	Key Mechanism
Fast-dLLM	Masked	$27.6\times$	Block-wise cache + parallel decode
dKV-Cache	Masked	$2.4\times$	Delayed post-decode caching
dLLM-Cache	Masked	$9.1\times$	V-verify cosine similarity
Elastic-Cache	Masked	$45.1\times$	Layer-aware adaptive refresh

Table 3.1: Existing KV caching methods for diffusion language models. All target masked diffusion.

The speedups in Table 3.1 are not directly comparable: methods are evaluated on different models (LLaDA-8B vs. LLaDA-1.5), tasks, and prompt and response lengths. Fast-dLLM’s $27.6\times$ also compounds caching with confidence-aware parallel decoding; its caching contribution alone is substantially lower. Our contribution is

demonstrating KV caching is viable for uniform diffusion, not a head-to-head systems comparison against masked-diffusion caching, which remains out of scope given these differences.

Chapter 4: Method

4.1 Problem Formulation

Consider a uniform diffusion language model with a context window of S positions and vocabulary size V . The context contains a prompt of S_p tokens followed by S_r response tokens being generated, with $S_p + S_r \leq S$ (in practice the prompt and response may be shorter than the context window, with remaining positions padded). The model uses block-wise denoising with block size B and T denoising steps per block. At each step t within a block, the model processes all S positions simultaneously, represented as $\mathbf{x}_t \in \{1, \dots, V\}^S$, and produces logits over the vocabulary for each position in the current block.

The uncached baseline forward pass costs $O(L \cdot (S^2 \cdot d + S \cdot d^2))$ per step, where L is the number of layers and d the hidden dimension: $S^2 \cdot d$ for bidirectional self-attention (S queries over S keys, cost d each) and $S \cdot d^2$ for the per-position linear projections (QKV, output, and MLP) applied to S positions. Block-only caching reduces the cost of each cached step within a block to $O(L \cdot (B \cdot S \cdot d + B \cdot d^2))$: only the B current-block positions are processed, so B queries attend over S cached KV entries and the linear projections scale with B instead of S . The first step of each block retains the full forward-pass cost to refresh the cache. The goal is to avoid redundant forward passes for positions whose KV representations have not meaningfully changed.

4.2 Three Generation Strategies

We define three generation strategies with increasing degrees of caching. Figure 4.1 shows which sequence regions are recomputed vs. reused at each step for all three strategies.

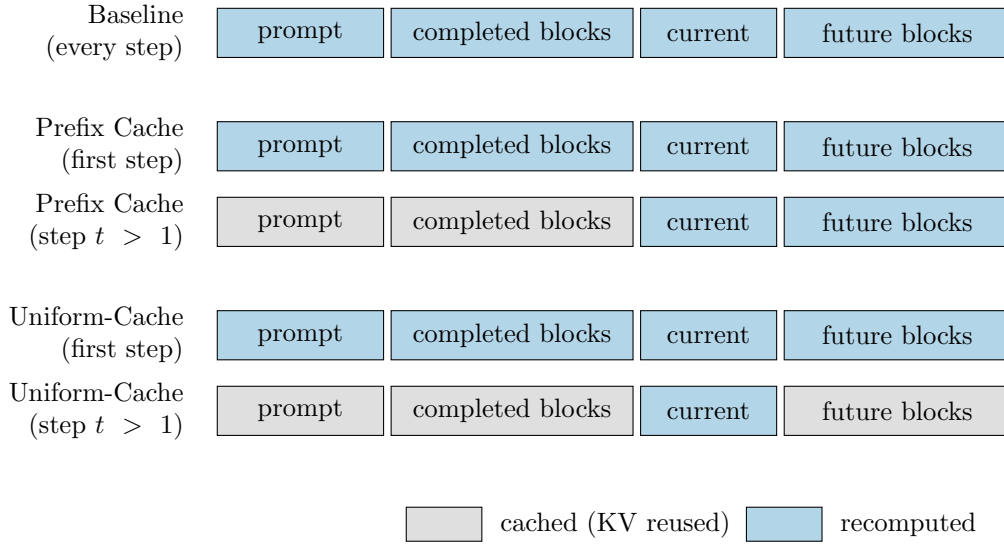


Figure 4.1: Sequence regions recomputed (blue) vs. cached (gray) per denoising step for each generation strategy. On the first step of each block all strategies run a full forward pass; strategies differ in what they cache on subsequent steps.

4.2.1 Baseline (No Cache)

Every step runs a full forward pass over all S positions. Nothing is reused across steps. This is the ground-truth reference. Total computation scales as $O(T_{\text{total}} \cdot S)$, where $T_{\text{total}} = T \cdot N_{\text{blocks}}$ is the total number of denoising steps across all blocks.

4.2.2 Prefix Cache

At the start of each block, a prefix forward pass computes KV states for all positions before the current block (prompt + completed blocks). Subsequent steps within that block reuse these prefix KV states and only recompute the current block and all positions ahead of it. The prefix grows by B with each completed block: it starts at S_p for the first block and reaches $S_p + S_r$ at the end of generation. The suffix recomputed each step averages $S - S_p - S_r/2 + B/2$ positions across all $N_{\text{blocks}} = S_r/B$ blocks. The speedup is therefore $S / (S - S_p - S_r/2 + B/2)$. When the response dominates ($S_r \approx S$, $S_p \approx 0$), this approaches $2\times$. When the prompt dominates

($S_p \gg S_r$), the suffix shrinks relative to S and the speedup can be substantially larger.

4.2.3 Uniform Cache

On the first step of each block, a full forward pass populates the entire KV cache. On subsequent steps, only the current block is recomputed: the block’s tokens are embedded and passed through all transformer layers, with each layer attending block queries against the full KV from cache. We also recompute the next block every r steps, where r is a hyperparameter. The block immediately ahead receives non-trivial attention from current-block queries and accumulates more drift than blocks further out; a periodic refresh keeps its KV states accurate without a full forward pass.

The theoretical speedup per block is $T \cdot S / (S + (T - 1)B)$, approaching S/B for large T , excluding the periodic next-block recomputes. At $S = 2048$, $B = T = 32$ this gives $\approx 21.6\times$; we achieve $12.2\times$ empirically. The gap reflects factors not captured by position counts alone: on each cached step, the full KV cache must still be read from memory to compute attention regardless of how few tokens are being recomputed, and the next-block recomputes add overhead not reflected in the formula.

4.3 Algorithm

Algorithm 1 Uniform-Cache: Block-Wise Cached Generation

Require: Model f_θ with L layers, prompt tokens $\mathbf{x}_{\text{prompt}}$, block size B , steps T , recompute interval r

- 1: Initialize KV cache $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{L \times \text{batch} \times S \times H \times D}$
- 2: Sample initial noised response $\mathbf{x}_0^{\text{resp}} \sim \text{Uniform}(\mathcal{V})^{S_r}$
- 3: **for** each block $b = 1, \dots, \lceil S_r/B \rceil$ **do**
- 4: Set block window: positions $[S_p + (b-1)B, S_p + bB)$ as current block \mathcal{B}
- 5: **for** each step $s = 1, \dots, T$ **do**
- 6: **if** $s = 1$ (first step of block) **then**
- 7: Run full forward pass over all S positions
- 8: Populate $\mathbf{K}[l], \mathbf{V}[l]$ for all layers l and all positions
- 9: **else**
- 10: Determine recompute region \mathcal{R} :
- 11: **if** $s \bmod r = 0$ and $b < \lceil S_r/B \rceil$ **then**
- 12: $\mathcal{R} \leftarrow \mathcal{B} \cup \mathcal{B}'$ where $\mathcal{B}' = \mathcal{B} + B$ $\{2B$ positions; refresh next block $\}$
- 13: **else**
- 14: $\mathcal{R} \leftarrow \mathcal{B}$ $\{\text{block-only recompute}\}$
- 15: **end if**
- 16: Embed tokens in \mathcal{R} and pass through all L layers
- 17: **for** each layer $l = 1, \dots, L$ **do**
- 18: Project Q/K/V for positions in \mathcal{R} ; apply RoPE
- 19: Update cache: $\mathbf{K}[l][\mathcal{R}] \leftarrow \hat{\mathbf{K}}, \mathbf{V}[l][\mathcal{R}] \leftarrow \hat{\mathbf{V}}$
- 20: Compute attention: queries for \mathcal{R} attend to full $\mathbf{K}[l], \mathbf{V}[l]$ (all S positions)
- 21: Apply FFN to hidden states of \mathcal{R}
- 22: **end for**
- 23: **end if**
- 24: Run LM head on \mathcal{B} outputs to get logits
- 25: Sample updated tokens from logits using selected sampling strategy
- 26: **end for**
- 27: Freeze block b : update noise mask to mark \mathcal{B} as completed
- 28: **end for**
- 29: **return** Generated response tokens

4.4 Design Decisions

The three design decisions below are empirically justified operating points supported by the drift and attention analyses in Chapter 6. Deriving a provably optimal refresh policy is left for future work.

4.4.1 Block-Only Recompute

During a block’s denoising steps, only the current block’s tokens change as sampling draws exclusively from the current block. However, because the model uses bidirectional attention, changes in the current block propagate through attention to all other positions, causing their KV projections to drift even though their token identities are fixed. Whether this drift is small enough to justify caching is not obvious a priori, and the answer depends on which region of the sequence we consider.

For completed blocks and the prompt, caching is safe *by construction*: GIDD’s structured attention mask prevents these positions from attending to the current block, so their representations are determined entirely by stable inputs. Their KV drift is exactly zero — not approximately zero, but structurally guaranteed. This is why prefix caching was baked into GIDD’s training from the start; no empirical justification was needed.

Future blocks are an entirely different case. They attend to everything, including the actively changing current block, with no structural constraint on their representations. The central finding that enables Uniform-Cache is empirical: our block drift analysis (Section 6.6.2) shows that future block KV drift is negligible in practice — the rest of future blocks (not counting the immediate next block) has $48\times$ lower drift than the current block — despite the absence of any architectural guarantee. The current block, which is actively being denoised and has tokens changing every step, has $24\times$ higher drift than even the immediately adjacent completed block. The attention mass directed at future blocks from current-block queries is also small: the rest of future blocks receive only 4.1% of attention mass. We therefore

treat future-block drift as negligible and do not refresh it between first-step forward passes — a heuristic supported by the empirical drift and attention numbers. In all cases, block queries attend to the full KV cache across all S positions, preserving complete bidirectional context.

4.4.2 Full Forward on First Step of Each Block

At each block boundary, the newly completed block is added to the completed set in the attention mask, changing attention patterns for all completed positions: they gain access to the new block, and the new block loses access to future blocks. This causes a guaranteed discontinuous shift in KV representations across all completed positions (visible in Figure 6.7). Future blocks also accumulate drift over the previous block’s denoising steps. To avoid carrying this discontinuity into subsequent cached steps, we adopt the conservative choice of running a full forward pass on the first step of each block, refreshing the entire cache before denoising begins; subsequent steps within the block are cached. More selective refresh strategies could reduce this cost, but GIDD’s asymmetric attention mask makes selective refresh of previous blocks difficult to implement cleanly, if not impossible. We leave this to future work (Section 7.3).

4.4.3 Recompute Interval for the Next Block

Every r steps the next block’s KV states are also refreshed by extending the recompute region from B to $2B$ positions; sampling still draws only from the current block. The block drift analysis shows the next block receives approximately 3.1% of attention mass from current-block queries and has moderate drift (0.008 cosine distance, compared to 0.048 for the current block). We choose $r = 4$ as a default operating point: often enough to keep the next block’s KV moderately fresh given its drift and attention mass, while keeping the per-step cost close to block-only recompute. The ablation in Section 6.7.2 shows that r trades off speedup against gen.

PPL; $r = 4$ is a reasonable compromise rather than a provably optimal value.

Blocks further ahead (“rest future”) receive 4.3% of attention mass in aggregate but have lower drift, so we do not refresh them during the generation of a block; their cached values remain accurate without intervention in our measurements.

4.4.4 Adaptive Sampling as Default

Two sampling strategies are supported:

Ancestral sampling follows the learned reverse diffusion process directly. At each of T pre-specified steps, all positions simultaneously sample from the model’s predicted denoising distribution $p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$, stepping from pure noise toward clean data. In our experiments, token change rate starts near 25% in the first steps and decays to below 2% for the remainder of generation (Figure 4.2).

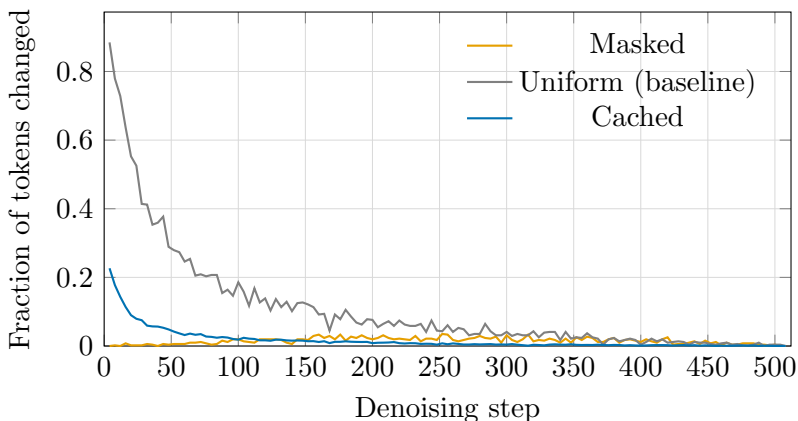


Figure 4.2: Token change rate per step under ancestral sampling for masked and uniform, with block size equal to the full response length. Rate starts near 25% and decays rapidly.

Adaptive sampling (von Rütte et al., 2025b) denoises k positions per step rather than all at once. At each step, positions are ranked by the following score and the top k are fully denoised to their argmax prediction; all others remain unchanged and the scores are recomputed on the next forward pass. We use $k = 3$, which we

found to work well in our ablation (Table 6.11); von Rütte et al. (2025b) used $k = 1$.

$$\text{score}(x_t) = \pi(x_t) \cdot \left(\max_{x'_0} p_\theta(x_0 = x'_0 \mid \mathbf{x}_t) - p_\theta(x_0 = x_t \mid \mathbf{x}_t) \right)$$

The first term, $\pi(x_t)$, measures how likely the current token is to be noise under the prior π ; the second measures how much the model expects to gain by changing it. Together, a high-scoring position is both likely to be corrupted and has a clear correction available. For masked diffusion, where π concentrates on [MASK], this reduces to selecting the most confident prediction among masked positions.

Block-wise generation is what enables caching: by processing the response in fixed windows, KV states for all completed regions can be reused across steps, recomputing only the current block. Adaptive sampling pairs naturally with this structure — its sparse, targeted updates keep cached KV states accurate throughout each block. Ancestral sampling samples from the learned reverse process $p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$ at each step, trained on full-sequence joint denoising; block-wise generation violates this training distribution, degrading quality independently of caching. Adaptive sampling is therefore our primary configuration; full ancestral results are in Appendix B.

Chapter 5: Experimental Setup

5.1 Models

All main experiments use the `dvruette/gidd-unif-3b` model (3B parameters, pure uniform diffusion) as the primary target. The `dvruette/gidd-unif-10b` model (10B parameters) is used in Section 6.8 to test whether the caching approach generalizes across model scales. For the token dynamics analysis (Section 6.5) and KV drift analysis (Section 6.6), we additionally compare against `dvruette/gidd-mask-3b` (3B parameters, masked diffusion) to contrast the caching properties of uniform vs. masked noise.

5.2 Hardware

Experiments were distributed across 4 NVIDIA A100 80GB PCIe GPUs via a parallel dispatcher; each individual experiment ran on a single GPU. The host machine has two Intel Xeon Silver 4314 CPUs (16 cores each, 2.40 GHz), 251 GB of system RAM, CUDA 12.8, PyTorch 2.10.0, and Transformers 4.57.6.

5.3 Datasets

Prompts are extracted from the WikiText-103 test set (Merity et al., 2017). Prompt length is 128 tokens for most experiments. The number of prompts and response length vary by experiment and are noted per-configuration in Section 5.6.

5.4 Evaluation Metrics

5.4.1 Quality: Generative Perplexity

We measure generation quality using generative perplexity, a standard metric in discrete diffusion evaluation (Lou et al., 2024; Sahoo et al., 2024; von Rütte

et al., 2025a): generated responses are scored under an external autoregressive judge, and perplexity $\text{PPL} = \exp(\text{mean NLL})$ is reported on the response portion. We use `nvidia/Minitron-4B-Base` (Sreenivas et al., 2024) as the judge, which shares the same training data distribution as the GIDD models (Nemotron-CC). We use it to compare generation quality between baseline and cached strategies on identical prompts. This captures one axis of generation quality under a single judge-dataset pair; whether the perplexity trends we observe generalize to other judges, datasets, or downstream tasks remains an open question.

5.4.2 Efficiency: Wall-Clock Time

End-to-end generation time is measured using CUDA events around full generation passes, without per-operation instrumentation overhead. A warmup pass is performed before timing to eliminate JIT compilation effects.

5.4.3 Throughput

Tokens per second are measured across batch sizes for the batching experiment, using dynamic batching via the `BatchScheduler` for the Uniform Cache strategy.

5.4.4 Operation-Level Profiling

Per-module CUDA timing via PyTorch forward hooks on model submodules (embedding, QKV projection, attention, MLP, layer norms, LM head). Both baseline and cached use identical hooks to ensure a consistent, like-for-like comparison. Operations not captured by hooks (RoPE, residuals, sampling, cache management) appear as “Other” via total-minus-hooked.

5.5 Hyperparameters

Unless otherwise specified, all experiments use:

Parameter	Value	Description
Response length (S_r)	512	Response tokens
Prompt length (S_p)	128	Prompt tokens
Sampling method	Adaptive	Top- k score-based updates
n_{tokens}	3	Tokens updated per step (adaptive)
Recompute interval (r)	4	Next-block refresh interval
Num. prompts	16–64	Varies by experiment (see Section 5.6)
Batch size	8	(benchmark); varies for batching experiment
Seed	42	Random seed

Table 5.1: Default hyperparameters for all experiments.

Two primary configurations are evaluated:

- **s32/b32**: 32 denoising steps per block, block size 32 (16 blocks). Primary configuration for end-to-end speedup evaluation. The 32-token block is 1.6% of the 2048-position context, so cached steps recompute a small fraction of the sequence.
- **s512/b512**: 512 denoising steps, block size 512 (single block spanning the full response). With no completed blocks, only the prompt and padding tokens contribute cached KV states.

5.6 Experiments

Benchmark End-to-end wall-clock timing and quality for all three strategies at s32/b32 and s512/b512 under both adaptive and ancestral sampling, plus a longer-response variant at s32/b32 adaptive (512 prompt and 1536 response filling the 2048 context with no padding), mimicking a realistic long-form setting where prior generation is re-fed as prompt to continue past the context window.

Benchmark Batching Throughput comparison (tokens/s, requests/s) across batch sizes 1, 2, 4, 8, 16 for the s32/b32 adaptive configuration.

Operation Profiling Per-module timing breakdown using PyTorch forward hooks with CUDA event timers for baseline vs. Uniform Cache at s32/b32 and s512/b512 (3B model only).

Step Degradation Quality comparison between masked, uniform, and cached models at s512/b512 with ancestral sampling, sweeping step counts from 512 down to 4 by halving (512, 256, ..., 4).

Token Dynamics Token change rates, entropy convergence, and max-probability trajectories for masked, uniform, and cached models.

KV Drift Analysis KV drift patterns across layers, timesteps, and sequence regions. Prompt vs. response drift, key vs. value drift, cross-layer correlation.

Block Drift KV drift and attention mass by block region throughout denoising.

Ablations Block size (b16/32/64/128/256), recompute interval ($r \in \{0, 1, 4\}$), and tokens per step ($n \in \{1, 3, 5, 10\}$).

10B Generalization Main benchmark on the 10B GIDD model at s32/b32 adaptive.

Chapter 6: Results

6.1 Main Benchmark

Table 6.1 presents the end-to-end benchmark results comparing all three strategies. Uniform-Cache achieves $12.0\times$ speedup at s32/b32 (r512), $11.6\times$ at s32/b32 (r1536), and $2.8\times$ at s512/b512 with adaptive sampling and a batch size of 8, while simultaneously reducing measured generative perplexity across all three configurations.

Config	Base (s)	Prefix (s)	Cached (s)	Base PPL	Prefix PPL	Cached PPL
s32/b32, r512	5,310	4,267 (1.24 \times)	441 (12.0\times)	5.33	5.03	4.86
s512/b512	5,311	4,962 (1.07 \times)	1,888 (2.8\times)	10.20	8.16	5.79
s32/b32, r1536 [†]	1,465	607 (2.42 \times)	126 (11.6\times)	3.74	3.91	2.98

Table 6.1: End-to-end benchmark results, all with adaptive sampling and batch size 8. r512 = 512 response tokens with 128-token prompt (64 prompts); r1536 = 1536 response tokens with 512-token prompt (including BOS token; 6 prompts). Speedup is relative to baseline. 95% bootstrap confidence intervals for cached PPL are reported in the text. [†]Only 6 prompts available in WikiText-103 with sufficient length (≥ 512 tokens including BOS) for the 1536-token response condition.

Several observations stand out:

- **s32/b32 benefits far more from caching than s512/b512.** In the s32/b32 configuration, the current block is 32 of 2048 total positions (1.6%), so cached steps recompute only 1.6% of the sequence. In the s512/b512 configuration, the single block spans 512 of 2048 positions (25%), leaving only the prompt and padding as cacheable context. This block-size dependence directly explains the $12.0\times$ vs. $2.8\times$ contrast.
- **Caching reduces generative perplexity under adaptive sampling.** Uniform-Cache reduces measured generative perplexity over the baseline across all configurations (s32/b32: 5.33 [4.79, 5.93] \rightarrow 4.86 [4.30, 5.49]; s512/b512: 10.20

[7.49, 14.33] \rightarrow 5.79 [4.87, 7.03]; r1536: 3.74 [3.10, 4.53] \rightarrow 2.98 [2.33, 3.77]; 95% bootstrap CIs). This consistent reduction is a surprising finding; we observe it robustly across tested configurations but do not claim a definitive causal account. PPL differences between baseline and prefix cache reflect a bfloat16 cuBLAS kernel artifact rather than true differences in sampling trajectory; see Appendix D for details. Ancestral results are omitted here for the reasons discussed in Section 4.4.4; see Appendix B for the full numbers.

- Prefix cache speedup scales primarily with prompt length.** At r512 (128-token prompt), the prefix cache achieves only 1.07–1.24 \times speedup: the 128-token prompt is only 6% of the 2048-token context, so caching it saves little per step. At r1536 (512-token prompt), the prompt is 25% of the context and the prefix cache reaches 2.42 \times speedup, with additional contribution from completed blocks accumulating over a longer generation. Response length alone has limited leverage for the prefix cache — even a response spanning the entire context could at most double the speedup — whereas prompt length directly determines the cacheable fraction. Uniform-Cache reaches 11.6–12.0 \times regardless of either, since it caches the entire sequence outside the current block rather than just the prompt and completed prefix.

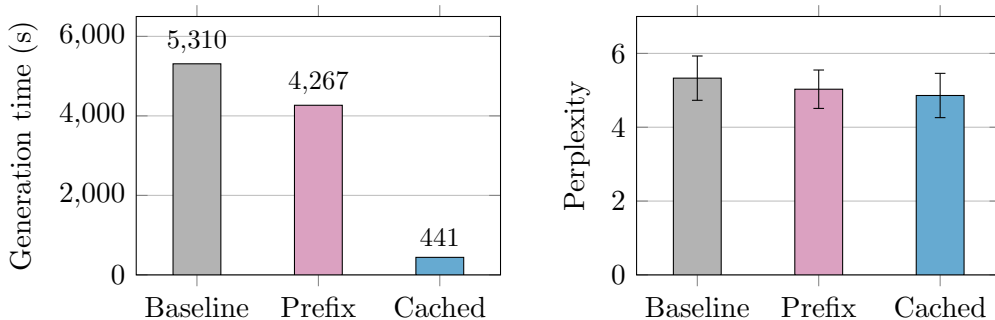


Figure 6.1: Generation time (left) and perplexity (right) for the three generation strategies at s32/b32 adaptive. Uniform-Cache is 12.0 \times faster than the baseline while also reducing generative perplexity (5.33 \rightarrow 4.86).

6.2 Operation-Level Profiling

Table 6.2 presents the per-operation CUDA timing breakdown for s32/b32 adaptive, revealing where speedup originates.

Operation	Baseline (ms)	Cached (ms)	Reduction
Attention	368,500	27,400	13.4×
MLP	111,300	6,400	17.3×
QKV projection	38,300	3,200	11.9×
LM head	30,500	500	61.2×
Layer norms	28,700	3,900	7.4×
O projection	13,000	1,200	11.1×
Other	68,000	25,100	2.7×
Total	659,100	67,900	9.71×

Table 6.2: Per-operation CUDA timing for s32/b32 adaptive. Core compute operations achieve 7–17× reductions. “Other” includes RoPE, cache management, and sampling overhead.

Attention and MLP together account for $\sim 73\%$ of baseline compute and are reduced by 14.2× combined (13.4× and 17.3× individually). Both scale with the number of positions processed: cached steps operate on only 32 query positions rather than 2048. The profiling total (9.71×) is lower than the wall-clock speedup (12.0×) due to GPU synchronization overhead from the hooks, which is proportionally heavier for the faster cached operations.

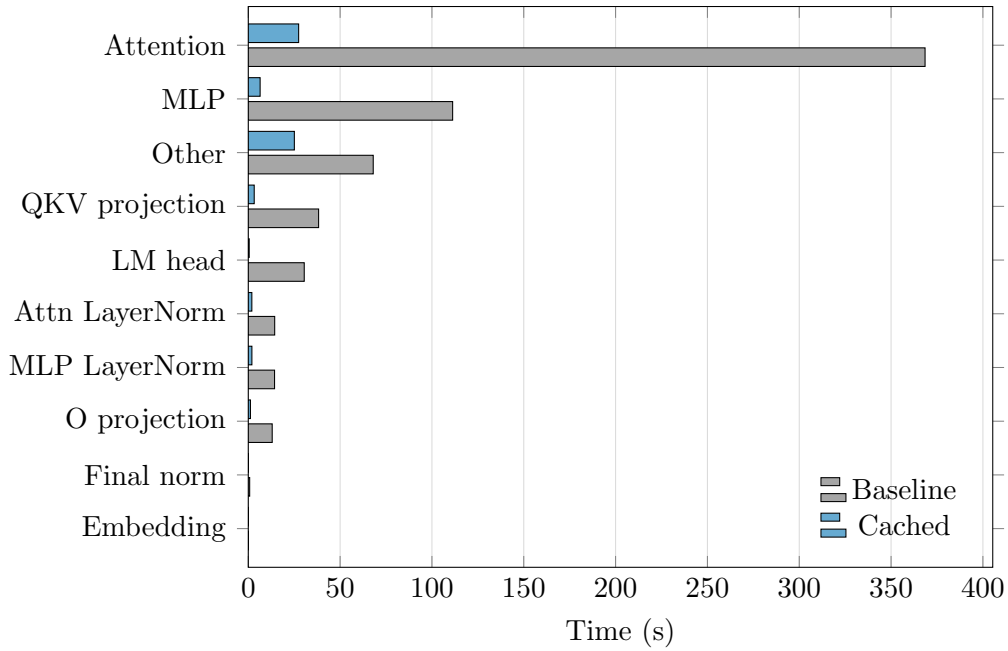


Figure 6.2: Per-operation CUDA timing breakdown for baseline vs. Uniform Cache at s32/b32. Attention dominates baseline compute (56% of total) and is reduced 13.4 \times by caching, but remains the largest single operation in the cached path.

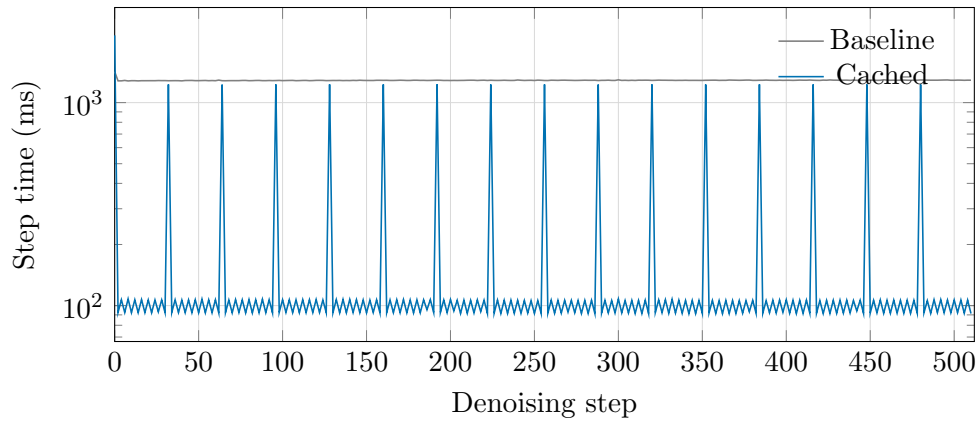


Figure 6.3: Per-operation CUDA time over denoising steps at s32/b32. Each data point is the cost of one operation type at a given step. First-of-block steps (full forward pass) appear as spikes in attention cost every 32 steps; between them, cached steps pay only block-level recompute cost.

6.3 Batching Throughput

The baseline is already compute-bound at batch size 1 — the full-sequence forward pass saturates the GPU, leaving no headroom for batch-level parallelism. Uniform Cache processes only 32-position blocks per step, leaving substantial GPU capacity that batch scaling exploits, reaching $2.4\times$ higher throughput at batch 16 than at batch 1.

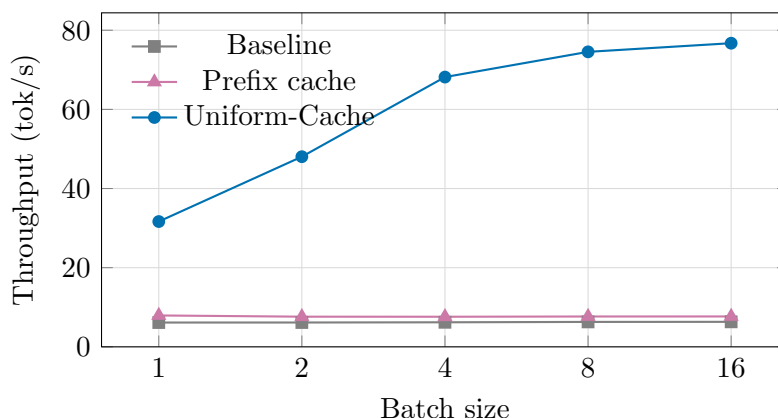


Figure 6.4: Token throughput (tok/s) vs. batch size for s32/b32 adaptive. Baseline is flat; Uniform Cache scales $2.4\times$ from batch 1 to 16.

6.4 Step Degradation

Figure 6.5 shows how quality degrades under ancestral sampling as the number of denoising steps decreases, comparing masked, uniform, and cached generation.

Two findings stand out. First, the cached and uncached uniform lines track each other closely at all step counts, confirming the caching approximation does not introduce additional degradation beyond the uncached baseline. Second, uniform diffusion degrades substantially more gracefully than masked diffusion as steps decrease. This is consistent with the self-correction advantage described in Section 2.2: masked diffusion’s curse of parallel decoding imposes a hard limit on progress per step — errors committed when unmasking multiple tokens simultaneously are permanent —

while uniform diffusion can revise any position at any subsequent step, recovering from errors introduced by aggressive step reduction.

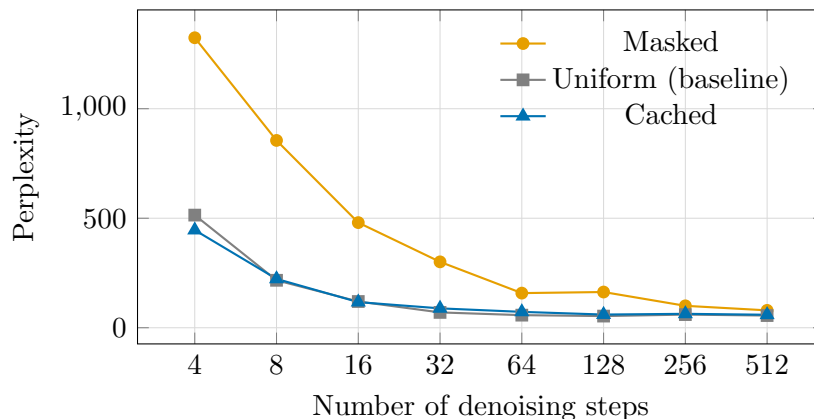


Figure 6.5: Perplexity vs. number of denoising steps for masked, uniform, and cached generation at s512/b512 under ancestral sampling. Cached and uniform track closely at all step counts. Uniform degrades substantially more gracefully than masked as steps decrease, consistent with uniform diffusion’s self-correction capability: errors from aggressive step reduction can be revised in subsequent steps, whereas masked diffusion’s permanently committed tokens cannot be undone.

6.5 Analysis of Token Dynamics

We characterize the empirical properties of uniform diffusion denoising that underpin our caching design.

6.5.1 Token Change Rates

With adaptive sampling, exactly $n_{\text{tokens}} = 3$ response tokens change per step by construction ($3/512 \approx 0.6\%$). With ancestral sampling, the change rate is model-determined and highly non-uniform: it starts near 25% in the first steps then decays rapidly, averaging 0.97–1.81% of response tokens per step across all steps (roughly 5–9× the masked rate). See Figure 4.2.

6.5.2 Entropy Convergence

Sampling	Masked	Uniform	Cached
Adaptive (start \rightarrow end)	7.21 \rightarrow 2.63	7.27 \rightarrow 0.48	7.27 \rightarrow 0.43
Ancestral (start \rightarrow end)	7.21 \rightarrow 2.89	7.27 \rightarrow 2.21	7.27 \rightarrow 2.20

Table 6.3: Mean prediction entropy (nats) at start and end of generation at s512/b512.

All models start at near-maximum entropy (~ 7.3 nats). With adaptive sampling, uniform and cached converge to near-zero entropy (0.43–0.48 nats) while masked plateaus at 2.63 nats. Masked diffusion’s absorbing state constrains n to at most B/T (block size divided by steps): each position can only be decoded once and is permanently fixed, so decoding more than one token per step on average exhausts the block before generation completes. Uniform diffusion supports a much higher n , allowing positions to be revised across steps; with more updates per step and iterative refinement, the model can drive entropy toward zero in a way masked diffusion cannot. With ancestral sampling, convergence is slower and final entropy remains higher (2.2–2.9 nats).

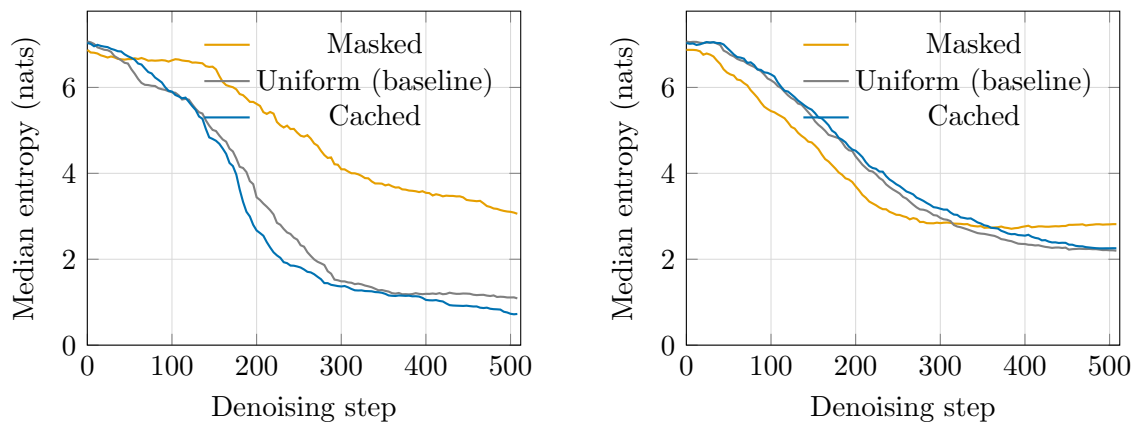


Figure 6.6: Per-token prediction entropy (nats) at s512/b512. *Left (adaptive)*: Uniform and cached converge to near-zero entropy while masked plateaus much higher. *Right (ancestral)*: Uniform and cached converge more slowly and to higher final entropy; masked converges comparably to the adaptive setting.

6.6 KV Drift Analysis

We measure KV drift as the cosine distance between a position’s key or value vectors at consecutive denoising steps. This section characterizes how drift is distributed across sequence regions, layers, and the relationship between key and value drift.

6.6.1 Key–Value Drift Correlation

Keys and values are separate projections and drift at different magnitudes. Table 6.4 shows that value drift is consistently 2–3× larger than key drift across all configs, models, and sampling strategies.

Config	Masked K/V	Uniform K/V	Cached K/V
s32 adaptive	0.0005 / 0.0012	0.0007 / 0.0023	0.0005 / 0.0013
s32 ancestral	0.0005 / 0.0014	0.0008 / 0.0024	0.0007 / 0.0018
s512 adaptive	0.0005 / 0.0012	0.0007 / 0.0021	0.0006 / 0.0020
s512 ancestral	0.0005 / 0.0013	0.0010 / 0.0029	0.0009 / 0.0025

Table 6.4: Mean key and value drift (cosine distance per step) across configs and models. Value drift is consistently 2–3× larger than key drift.

Despite the magnitude difference, the question for caching is whether they drift in the same *places*. Table 6.5 shows the global Pearson correlation between key and value drift magnitudes.

Model	Global $r(\mathbf{K}, \mathbf{V})$
Masked	0.94
Uniform	0.93
Cached	0.87

Table 6.5: Pearson correlation between key and value drift magnitudes at s32/b32 adaptive.

Key and value drift are highly correlated across all three models ($r \geq 0.87$):

positions with high key drift also have high value drift. Both projections respond to token changes in the same locations, so either metric is representative of where drift is concentrated.

6.6.2 Drift by Block Region

Table 6.6 shows average KV drift by sequence region.

Region	s32/b32 adaptive	s32/b32 ancestral
Prompt	0.000042	0.00005
Rest past	0.00016	0.0002
Previous block	0.0020	0.003
Current block	0.050	0.063
Next block	0.0075	0.003
Rest future	0.0012	0.0007
Padding	0.00063	—

Table 6.6: Mean cosine-distance KV drift (average of key and value) per step by sequence region at s32/b32.

Drift is sharply concentrated in the current block. The immediately preceding block has $24\times$ less drift than the current block; all other completed blocks (“rest past”) have $240\times$ less drift; and the prompt has $1000\times$ less drift. Even the worst-case completed block is an order of magnitude lower than the current block, directly justifying the block-only recompute design.

The next block has moderate drift, about $6\times$ less than the current block but $4\times$ more than completed blocks. This is the rationale for the recompute interval: periodically refreshing the next block’s KV (every 4 steps) prevents accumulation of moderate drift without paying the cost of refreshing it every step.

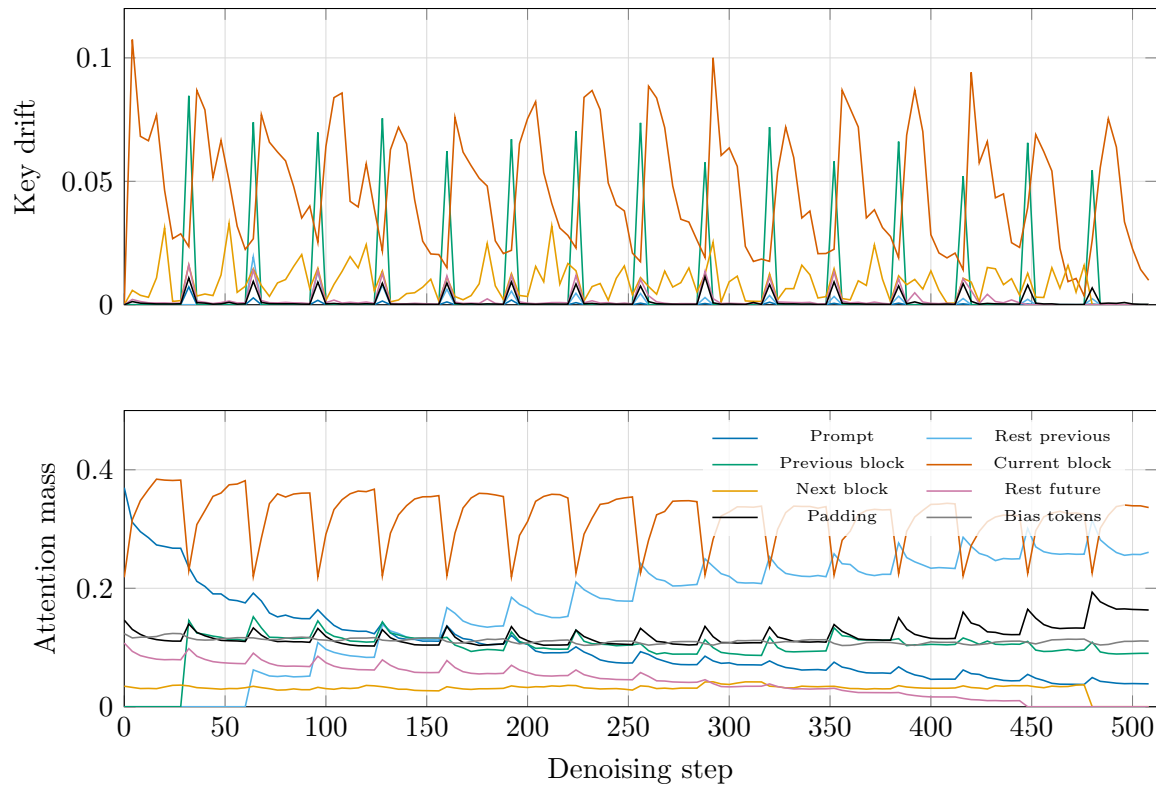


Figure 6.7: Top: mean KV drift (average of key and value) by sequence region over all denoising steps at s32/b32. The current block dominates; completed blocks and prompt are orders of magnitude lower. Bottom: attention mass from current-block queries to each region. Completed blocks and prompt receive $\sim 27\%$ and $\sim 10\%$ of attention respectively despite near-zero drift.

6.6.3 Attention Mass by Region

Table 6.7 shows attention mass by region for both configurations; see also Figure 6.7 (bottom panel) for the s32/b32 breakdown over time.

Region	s32/b32	s512/b512
Bias token	11.1%	11.0%
Prompt	10.2%	8.6%
Rest past	16.6%	—
Previous block	9.8%	—
Current block	32.9%	68.0%
Next block	3.1%	—
Rest future	4.3%	—
Padding	11.9%	12.3%

Table 6.7: Fraction of attention mass directed by current-block queries to each sequence region.

6.6.4 Cross-Layer Drift Correlation

Config	Masked		Uniform		Cached	
	K	V	K	V	K	V
s32/b32 adaptive	0.90	0.90	0.81	0.79	0.83	0.85
s512/b512 adaptive	0.94	0.94	0.79	0.80	0.83	0.86
s512/b512 ancestral	0.90	0.90	0.73	0.76	0.75	0.76

Table 6.8: Mean off-diagonal cross-layer drift correlation (K = keys, V = values) across configs and models.

Masked diffusion shows high cross-layer correlation (0.90–0.94), meaning all layers behave similarly. Uniform diffusion shows lower correlation (0.74–0.83), meaning drift varies more across layers. We explored a per-layer threshold-based alternative that refreshed only positions whose KV drift exceeded a set threshold, but noncontiguous memory accesses and KV drift check overhead made it slower in practice than the simpler block-only recompute, despite reducing the number of positions updated. A more hardware-efficient implementation could make this finer-grained approach viable and represents a potential direction for future work.

6.7 Ablations

6.7.1 Block Size

Block size	Base PPL	Cached PPL	Speedup
16	6.09 [4.60, 8.06]	6.19 [4.69, 7.95]	9.5×
32	5.33 [4.79, 5.93]	4.86 [4.30, 5.49]	12.0×
64	3.54 [2.63, 4.64]	3.84 [3.07, 4.73]	11.3×
128	4.51 [3.38, 5.94]	3.73 [3.06, 4.49]	8.4×
256	6.27 [4.70, 8.30]	4.11 [2.90, 5.90]	5.1×

Table 6.9: Block size ablation with steps = block size, adaptive sampling (16 prompts). PPL shown with 95% confidence intervals.

Speedup peaks at b32 (12.0×) and falls for both smaller and larger block sizes. Generative perplexity under caching is generally comparable across configurations, with b32 and b64 offering the best combination of speedup and PPL.

6.7.2 Recompute Interval

Interval r	Base PPL	Cached PPL	Speedup
1 (every step)	5.42 [4.34, 6.65]	4.26 [3.66, 5.02]	10.6×
4 (default)	5.42 [4.34, 6.65]	4.69 [3.72, 5.87]	11.9×
0 (never refresh)	5.42 [4.34, 6.65]	4.79 [3.93, 5.79]	12.2×

Table 6.10: Recompute interval ablation at s32/b32 adaptive (16 prompts).

$r = 1$ (refreshing the next block every step) gives the lowest gen. PPL but reduces speedup to 10.6×. $r = 0$ (never refreshing) still reduces gen. PPL over baseline, showing that block-only caching accounts for most of the gen. PPL benefit. $r = 4$ is our chosen default, recovering near-peak speedup while retaining strong gen. PPL reduction.

6.7.3 Tokens per Step (n_{tokens})

n	Base PPL	Cached PPL	Speedup
3	5.42 [4.34, 6.65]	4.69 [3.72, 5.87]	11.8×
5	4.88 [3.89, 6.06]	5.93 [4.99, 7.07]	11.8×
10	7.27 [5.56, 9.43]	8.21 [6.80, 9.96]	11.8×

Table 6.11: Ablation on tokens decoded per step (n_{tokens}) at s32/b32 adaptive (16 prompts). $n = 1$ is omitted as a degenerate case: with only 32 total token updates for a 32-token block, base PPL is 234.2 and cached PPL is 371.3.

Generative perplexity is broadly similar across configurations, with $n = 3$ offering the best quality. Speedup is nearly constant across n since cached computation scales with block size, not the number of tokens updated per step.

6.8 Generalization to 10B Model

Model	Base (s)	Prefix (s)	Cached (s)	Base PPL	Cached PPL
GIDD 3B	5,310	4,267 (1.24×	441 (12.0 ×	5.33	4.86
GIDD 10B	3,789	3,080 (1.23×	314 (12.1 ×	4.74	3.64

Table 6.12: Benchmark results for GIDD 3B and 10B at s32/b32 adaptive. The 10B run uses 16 prompts; the 3B run uses 64. Speedup and gen. PPL reduction are consistent across model sizes.

Uniform-Cache generalizes to the 10B GIDD model with similar characteristics to the 3B results: 12.1× wall-clock speedup and a generative perplexity reduction under caching (4.74 → 3.64). The consistent speedup across model sizes shows that the caching properties observed at 3B are not specific to that scale.

Chapter 7: Conclusion

7.1 Summary of Contributions

This thesis showed that KV caching is viable for uniform diffusion language models. Token changes are sparse under both adaptive and ancestral sampling, and KV drift is semi-local. Block-wise generation clusters active changes within a single block, concentrating drift there and leaving all other regions — completed blocks, the prompt, and padding — with near-zero drift, making block-only recompute a natural fit with negligible impact on measured generative perplexity.

von Rütte et al. (2025b) showed that the quality gap between masked and uniform diffusion narrows as compute scales. The inference gap appeared to be a separate, structural disadvantage — masked diffusion’s absorbing state provides a natural caching signal that uniform diffusion lacks. This work closes that gap empirically by demonstrating that caching is viable for uniform diffusion and achieves strong speedups in our evaluation setup; a head-to-head comparison against masked-diffusion caching methods under matched conditions is outside our scope.

Our contributions are:

1. **Uniform-Cache, the first inference-time KV caching method for uniform diffusion language models.** A block-wise design that achieves up to $12.2\times$ speedup (s32/b32) over the uncached baseline, using the same sampling method and consistently reducing measured generative perplexity across all tested configurations.
2. **An empirical analysis of uniform diffusion denoising dynamics** that justifies block-only caching: token changes are sparse, KV drift follows token changes semi-locally, and drift is sharply concentrated in the current block. These observations explain why reusing cached KV for all non-current-block

regions introduces negligible error, and provide a foundation for future caching work on uniform diffusion.

7.2 Limitations

- **Evaluation limited to generative perplexity.** GIDD is a base pretrained model with no instruction fine-tuning, making standard reasoning benchmarks uninformative at this stage. Generative perplexity with an external judge is the appropriate standard for base diffusion models, but as uniform diffusion models are developed with post-training, benchmark-based evaluation of caching impact on generation quality should become the standard.
- **Single model family.** All experiments use GIDD (3B and 10B) because it is the only large uniform diffusion model currently available. The drift evidence — specifically that future blocks drift little despite GIDD’s mask giving them no special treatment — suggests the caching approach does not depend on GIDD-specific architectural features, but this is not directly verified.
- **Single dataset.** All experiments use WikiText-103 prompts. Whether the caching properties and gen. PPL results generalize to other domains or longer-form generation is untested.
- **Gen. PPL reduction is not fully explained.** Caching consistently reduces measured generative perplexity across all tested configurations, but we do not have a complete mechanistic account of why. We claim the robustness of the effect across configurations, not its cause.
- **No head-to-head comparison against masked-diffusion caching.** The speedup numbers reported for prior masked-diffusion caching methods are on different models, tasks, and sequence lengths (Table 3.1), and the masked and uniform settings differ structurally through the presence or absence of an absorbing state. We therefore do not position Uniform-Cache against those systems

under matched conditions. The contribution is establishing viability for uniform diffusion, not claiming superiority over masked-diffusion caching systems.

7.3 Future Work

- **Training without the structured attention mask.** GIDD’s baked-in attention mask forces a full forward pass at every block boundary and prevents global corrections across blocks. The empirical drift data suggests neither constraint is necessary for block-only caching to be safe. A model trained without this mask could revise any position at block boundaries, recovering the cross-block self-correction that uniform diffusion is theoretically capable of, while still benefiting from block-wise caching. More broadly, this provides suggestive evidence that low non-current-block drift is a property of block-wise uniform diffusion rather than merely an artifact of GIDD’s mask, and that Uniform-Cache may port to future uniform diffusion models trained without it; GIDD is currently the only large uniform diffusion model available, so this remains untested.
- **Combining with other acceleration methods.** Our speedup comes entirely from caching, using the same sampling method as the uncached baseline. Other acceleration approaches are orthogonal to caching and could yield multiplicative gains.
- **Hardware-optimized implementation.** The current implementation uses standard PyTorch operations. Custom CUDA kernels for the block-only attention pattern could close the gap between the theoretical $\sim 21.6\times$ position-count speedup and the empirical $12.2\times$.
- **Caching for self-correction.** The GIDD self-correction procedure (von Rütte et al., 2025a) iteratively corrects one token at a time over a completed generation — sparse token changes over mostly-stable context. Applying block-

only caching here could substantially reduce per-iteration cost, making post-generation self-correction more practical.

Appendix A: Intra-Block Denoising Dynamics

A.1 Entropy Convergence Within Blocks (s32/b32)

Figure A.1 shows the current block’s entropy over all denoising steps at s32/b32 for baseline and cached generation. The sawtooth pattern reflects the block-wise denoising process: entropy resets at the start of each block then converges as the block is denoised.

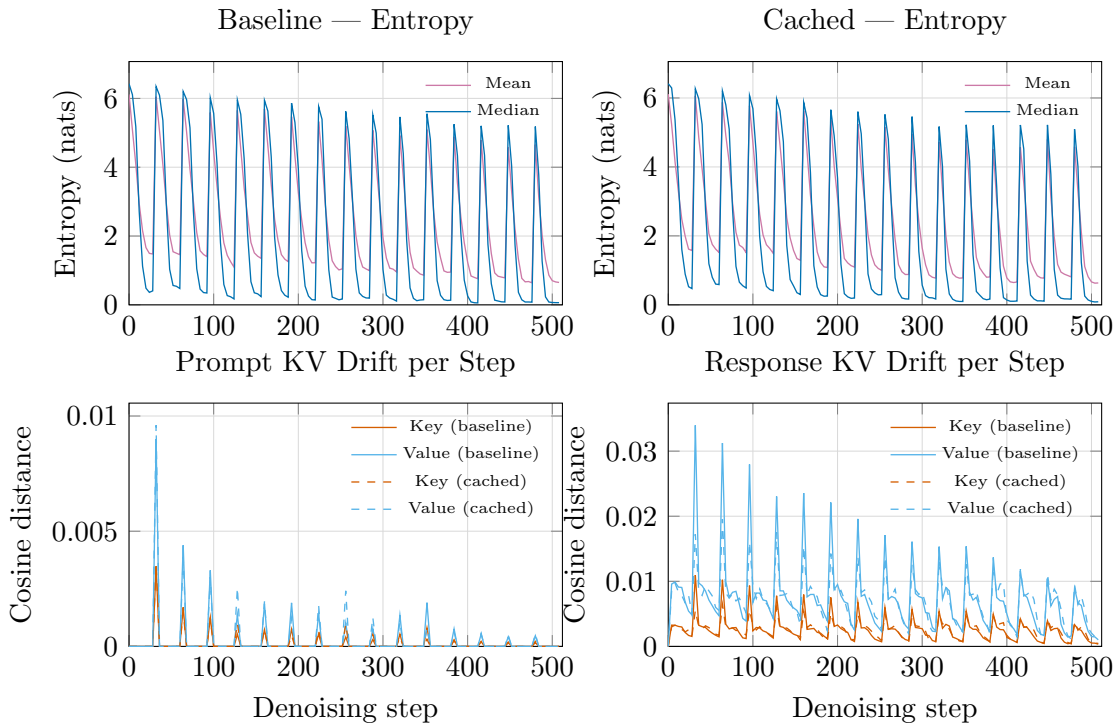


Figure A.1: Entropy convergence (top) and per-step KV drift (bottom) at s32/b32 adaptive sampling. Bottom panels show key and value drift separately for prompt positions (left) and response positions (right), comparing baseline and cached generation. The sawtooth entropy pattern reflects block-wise generation; later blocks start at lower entropy as the model conditions on more completed context.

A.2 KV Drift by Denoising Timestep (s32/b32)

Figure A.2 shows the current block’s average KV drift across denoising timesteps. Drift is highest at the beginning of each block (when the noise level is highest and the model is making large token updates) and falls as the block converges toward its final tokens.

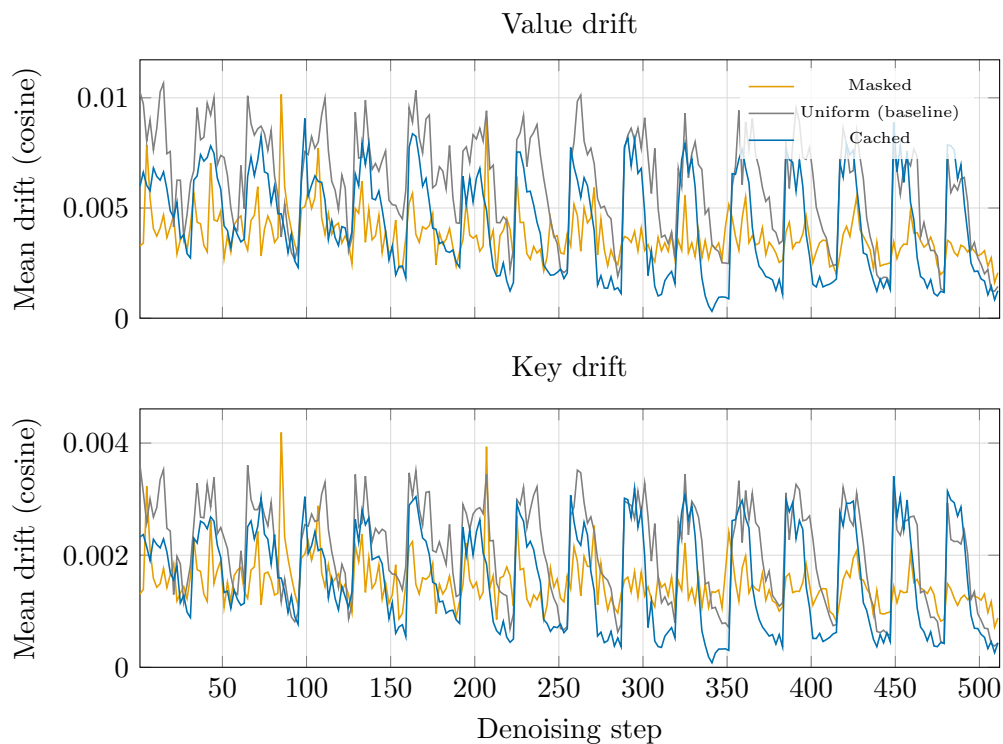


Figure A.2: Mean KV drift across denoising timesteps at s32/b32. Drift peaks at the start of each block (high noise) and decays as the block converges, directly mirroring the entropy convergence pattern.

Appendix B: Ancestral Sampling with Block-Wise Generation

As discussed in Section 4.4.4, block-wise generation violates the training distribution of ancestral sampling. As shown in Table B.1, the uncached ancestral baseline at s32/b32 already achieves PPL 179, far worse than s512/b512 (PPL 51.4), confirming that the primary degradation comes from block structure itself, not caching.

Config	Base (s)	Prefix (s)	Cached (s)	Base PPL	Cached PPL
s32/b32	1,303	1,069 (1.22 \times)	111 (11.7 \times)	179	214
s512/b512	1,351	1,238 (1.09 \times)	485 (2.8 \times)	51.4	56.4

Table B.1: Ancestral sampling benchmark results (ancestral sampling, batch size 8). Speedup is relative to baseline.

Caching introduces additional degradation at s512/b512 (51.4 \rightarrow 56.4) that does not appear under adaptive sampling. The likely cause is token change rate: with adaptive sampling, exactly $n_{\text{tokens}} = 3$ positions change per step, leaving the vast majority of KV states unaffected. With ancestral sampling, the change rate is higher and variable (roughly 1–2% of positions per step vs. \sim 0.6% for adaptive), so more positions accumulate stale KV states over a block.

Appendix C: Sample Generation Outputs

The following are sample outputs from Uniform-Cache at s32/b32 adaptive sampling. Prompts and responses are truncated for display.

Example 1

Prompt: The An Lushan Rebellion began in December 755, and was not completely suppressed for almost eight years. It caused e. . .

Response: written about it. Although there is a reliable source for the anthology, there is some consensus on Du Fu’s memorial, on Du Fu’s poetry. Du Fu’s poems express his love of the countryside and. . .

Example 2

Prompt: The single made its Irish Singles Chart debut at number 24 on the week ending 13 December 2012. It peaked at numbe. . .

Response: “Kiss You” peaked in the UK Singles Chart in the week ending 17 January 2013, marking their sixth top ten appearance in the United Kingdom. In the United States, “Kiss You” peaked at numb. . .

Example 3

Prompt: The Ise-class ships had two sets of direct-drive steam turbines, each of which drove two propeller shafts with. . .

Response: but depending on the boiler and the speed of the ship, the turbines could produce between 35,000 and 45,000 shaft horsepower (30,000 or 34,000 kW) or more, depending o. . .

Example 4

Prompt: A 70-metre-long (229 ft 8 in) flight deck was built above the stern and stretched forward to the rebuilt a . . .

Response: fitted to the flight deck just forward of the shooting range. Each catapult was fitted with a pair of guns, the calibre of the shells to be fired. A pair of 38 . . .

Appendix D: bfloat16 Numerical Artifact: Baseline vs. Prefix Cache PPL Gap

Baseline and prefix cache generation are mathematically equivalent: both compute the same attention over the same tokens, differing only in whether the prompt KV states are recomputed at every step or cached. In fp32 they produce bit-identical outputs. In bfloat16, however, they consistently produce slightly different perplexity values (e.g., 5.33 vs. 5.03 at s32/b32).

The source is cuBLAS GEMM kernel selection. The `down_proj` MLP matmul selects a kernel based on the input sequence length (the M dimension). Prefix caching changes the sequence length passed to the suffix forward pass, causing cuBLAS to select a different kernel with a different floating-point accumulation order. This introduces a per-layer difference of up to ~ 0.1 in bfloat16, which compounds across 19 layers and is large enough to perturb adaptive sampling decisions at block boundaries — producing different token sequences and hence different perplexity.

The same artifact affects Uniform-Cache PPL comparisons against baseline. All three strategies draw from the same generative distribution; PPL differences between strategies reflect sampling-trajectory sensitivity to bfloat16 accumulation order, not true quality differences. Running in fp32 reduces per-layer divergence by four orders of magnitude and collapses the PPL gap.

Works Cited

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Andrew Campbell, Joe Benton, Valentin De Bortoli, Tom Rainforth, George Deligiannidis, and Arnaud Doucet. A continuous time framework for discrete denoising models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Google DeepMind. Gemini Diffusion. <https://deepmind.google/models/gemini-diffusion/>, 2025.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Emiel Hoogeboom, Didrik Nielsen, Priyank Jaini, Patrick Forré, and Max Welling. Argmax flows and multinomial diffusion: Learning categorical distributions. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Inception Labs. Introducing Mercury, the world’s first commercial-scale diffusion large language model. <https://www.inceptionlabs.ai/blog/introducing-mercury>, 2025.

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.

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

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

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations (ICLR)*, 2017.

Quan Nguyen-Tri, Mukul Ranjan, and Zhiqiang Shen. Attention is all you need for KV cache in diffusion LLMs. *arXiv preprint arXiv:2510.14973*, 2025. Introduces the Elastic-Cache method.

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.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Technical Report*, 2019.

Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T. Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Yang Song, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. In *International Conference on Learning Representations (ICLR)*, 2021.

Sharath Turuvekere Sreenivas, Saurav Muralidharan, Raviraj Joshi, Marcin Chochowski, Ameya Sunil Mahabaleshwarkar, Gerald Shen, Jiaqi Zeng, Zijia Chen, Yoshi Suhara, Shizhe Diao, Chenhan Yu, Wei-Chun Chen, Hayley Ross, Oluwatobi Olabiyi, Ashwath Aithal, Oleksii Kuchaiev, Daniel Korzekwa, Pavlo Molchanov, Mostofa Patwary, Mohammad Shoeybi, Jan Kautz, and Bryan Catanzaro. LLM pruning and distillation in practice: The Minitron approach, 2024.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.

Dimitri von Rütte, Janis Fluri, Yuhui Ding, Antonio Orvieto, Bernhard Schölkopf, and Thomas Hofmann. Generalized interpolating discrete diffusion. In *International Conference on Machine Learning (ICML)*, 2025a.

Dimitri von Rütte, Janis Fluri, Omead Pooladzandi, Bernhard Schölkopf, Thomas Hofmann, and Antonio Orvieto. Scaling behavior of discrete diffusion language models. *arXiv preprint arXiv:2512.10858*, 2025b.

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

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.