dLLM-Cache: Accelerating Diffusion Large Language Models with Adaptive Caching

Zhiyuan Liu[1,2*] **Yicun Yang**[1,2*] **Yaojie Zhang**[2,3] **Junjie Chen**[2]
Chang Zou[1,2,3] **Qingyan Wei**[2] **Shaobo Wang**[1,2] **Linfeng Zhang**[1,2†]
[1]School of Artificial Intelligence, Shanghai Jiao Tong University
[2]EPIC Lab, Shanghai Jiao Tong University
[3]University of Electronic Science and Technology of China
{tpssplzy,yangyicun187}@gmail.com

Codes: https://github.com/maomaocun/dLLM-Cache

Abstract

Autoregressive Models (ARMs) have long dominated the landscape of Large Language Models. Recently, a new paradigm has emerged in the form of diffusion-based Large Language Models (dLLMs), which generate text by iteratively denoising masked segments. This approach has shown significant advantages and potential. However, dLLMs suffer from high inference latency. Traditional ARM acceleration techniques, such as Key-Value caching, are incompatible with dLLMs due to their bidirectional attention mechanism. To address this specific challenge, our work begins with a key observation that dLLM inference involves a static prompt and a partially dynamic response, where most tokens remain stable across adjacent denoising steps. Based on this, we propose dLLM-Cache, a training-free adaptive caching framework that combines long-interval prompt caching with partial response updates guided by feature similarity. This design enables efficient reuse of intermediate computations without compromising model performance. Extensive experiments on representative dLLMs, including LLaDA 8B and Dream 7B, show that dLLM-Cache achieves up to *9.1*$\times$ speedup over standard inference without compromising output quality. Notably, our method brings dLLM inference latency close to that of ARMs under many settings.

1 Introduction

Large language models (LLMs) [Zhao et al., 2023] are foundational to modern AI, powering applications from conversational AI to scientific discovery. While autoregressive models (ARMs) have been the dominant paradigm [Radford, 2018, Radford et al., 2019, Brown, 2020, OpenAI, 2022], diffusion-based large language models (dLLMs), such as LLaDA [Nie et al., 2025] and Dream [Ye et al., 2025], have emerged as promising alternatives. These models offer impressive scalability and, in some cases, outperform ARMs in handling challenges like the "reversal curse" [Berglund et al., 2023], demonstrating the potential of diffusion models for complex language tasks.

However, the practical adoption of dLLMs is hindered by their high computational demands during inference. Unlike ARMs, dLLMs require an iterative, multi-step denoising process, resulting in significant latency. What's worse, traditional ARM acceleration techniques, such as Key-Value caching [Pope et al., 2023], are incompatible with dLLMs due to their reliance on bidirectional attention. Furthermore, existing acceleration techniques for general diffusion models, primarily

[*]Equal contribution. [†]Corresponding author.

Preprint. Under review.

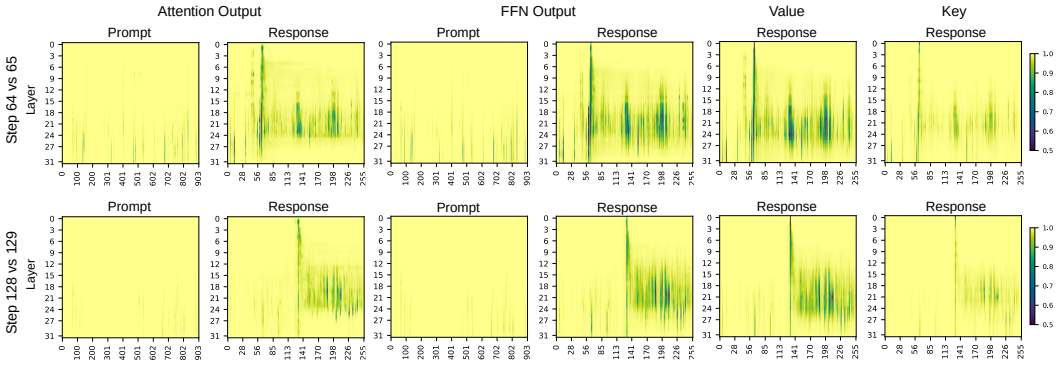


Figure 1: **Cosine similarity of Key, Value, Attention Output and FFN Output between two adjacent denoising steps in a dLLM**, highlighting computational redundancies. The heatmaps show similarity across adjacent steps for prompt and response tokens, respectively, where a lighter color indicates a higher similarity of a token compared with its value in the last step. These results demonstrate: (I) The prompt region exhibits high similarity, while the response region shows different similarity in different tokens. (II) Notably, only a small fraction of response tokens exhibit significantly lower similarity, suggesting that selective recomputation is sufficient. (III) Response tokens' value similarity closely aligns with attention and FFN output similarity, supporting that value changes can serve as an effective indicator to identify those most changed response tokens.

developed for vision, fall short when applied to dLLMs. General methods include reducing sampling steps, which has been proven in LLaDA to degrade performance, or accelerating the denoising network via feature caching [Ma et al., 2024, Selvaraju et al., 2024, Zou et al., 2024a,b, Liu et al., 2025]. However, these existing feature caching strategies typically apply a uniform policy to all tokens, overlooking the unique challenges of language tasks.

To solve this problem, we first study two computational redundancies in the inference process of dLLMs as illustrated in Figure 1, which uniform strategies fail to address. First, **prompt redundancy** arises because the input prompt tokens remain constant, yet their internal representations, *e.g.*, attention output, are recomputed in each denoising step. Second, **response dynamics and redundancy** occur as the generated response features evolve. While significant similarity often exists between adjacent steps, suggesting caching potential, not all tokens evolve in the same way. This non-uniform evolution explains why traditional uniform caching strategies are ineffective.

Motivated by these insights, we introduce **dLLM-Cache**, a training-free, adaptive caching mechanism designed to accelerate dLLM inference by exploiting these distinct redundancies. dLLM-Cache employs a differentiated caching strategy comprising two core components:

- **Long-Interval Prompt Caching:** We compute and cache features related to the prompt tokens only at sparse, long intervals, *e.g.*, every 100 steps. These cached features are then reused in all subsequent intermediate steps until the next long interval, drastically reducing the overhead associated with processing the static prompt.
- **Adaptive Short-Interval Response Caching:** Features associated with the response tokens are cached and fully refreshed at more frequent, shorter intervals, *e.g.*, every 10 steps. Between these full refreshes, we adopt an *adaptive partial update* strategy to balance speed and accuracy. Specifically, we identify and selectively update only the most dynamic tokens. As shown in Figure 1, the cosine similarity of a token's Value vector across adjacent steps strongly correlates with changes in its subsequent Attention and FFN Output. This motivates our **V-verify** mechanism, which uses Value similarity as an efficient proxy to select tokens for update.

This differentiated adaptive handling of prompt and response features allows significant inference acceleration while preserving quality, all without retraining. Our main contributions are:

1. We identify and characterize the dual computational redundancies in dLLM inference: quasi-static prompt and dynamic response redundancy.
2. We propose **dLLM-Cache**, a training-free adaptive caching framework that combines long-interval prompt caching with short-interval, similarity-guided partial updates for response tokens.

3. We introduce **V-verify**, a lightweight yet effective mechanism that uses cosine similarity of Value vectors across denoising steps to identify the most changed tokens for partial update, grounded in strong empirical correlation with overall token evolution.
4. We experimentally validate dLLM-Cache across various benchmarks, showing significant inference acceleration, *e.g.*, up to ***9.1***$\times$ on LLaDA with **lossless impact on response quality**, achieving a superior speed-quality trade-off compared to the baseline and simpler caching methods.

2 Related Work

2.1 The Landscape of Large Language Models

Autoregressive Models. Autoregressive model (ARM) is the dominant paradigm for large language models (LLMs), generating text sequentially via causal attention. These models underpin many state-of-the-art systems [Radford, 2018, Radford et al., 2019, Brown, 2020, OpenAI, 2022].

Diffusion Models for Language. Diffusion Models (DMs) [Sohl-Dickstein et al., 2015, Ho et al., 2020, Song et al., 2021] learn to reverse a data corruption process, excelling in continuous domains like images [Rombach et al., 2022, Peebles and Xie, 2023]. However, adapting DMs to discrete data like text presents unique challenges, partly due to text's discrete nature. A promising direction involves Masked Diffusion Models (MDMs) [Austin et al., 2021, Lou et al., 2023, Shi et al., 2024, Ou et al., 2024], a specific instance of discrete diffusion which operates on discrete sequences by iteratively predicting masked tokens based on their context.

Recent efforts have focused on scaling these MDMs significantly, positioning them as potential challengers to dominant ARMs. Notable examples include LLaDA [Nie et al., 2025], an 8B MDM trained from scratch using a bidirectional Transformer, and Dream [Ye et al., 2025], which initializes from pre-trained ARM weights. Both demonstrate performance comparable to similarly-sized ARMs like LLaMA3 8B [Dubey et al., 2024]. Their inherent bidirectional architecture may circumvent ARM limitations like the reversal curse [Berglund et al., 2023], making masked diffusion a competitive alternative for building foundation LLMs.

2.2 Acceleration via Caching Mechanisms

Key-Value Caching for ARMs. Key-Value caching [Pope et al., 2023] accelerates ARMs by reusing previous tokens' attention states, but is incompatible with dLLMs' bidirectional attention architecture.

Feature Caching for DMs. Feature caching in DMs leverages temporal redundancy by reusing intermediate features across denoising steps. The initial approaches focused on U-Net architectures, with DeepCache [Ma et al., 2024] and Faster Diffusion [Li et al., 2023] achieving early success. Following the emergence of Diffusion Transformers [Peebles and Xie, 2023], several caching paradigms have been developed. FORA [Selvaraju et al., 2024] employs static-interval caching for attention and MLP layers. While the ToCa series [Zou et al., 2024a,b] reduces information loss by dynamically updating cached features. More innovatively, TaylorSeer [Liu et al., 2025] introduces a *"cache-then-forecast"* paradigm that predicts features using Taylor series expansion rather than direct reuse.

Limitations of Existing Caching for dLLMs. Although diffusion acceleration has progressed, existing feature caching methods are ineffective for dLLMs. Based on our key observation that language tasks exhibit a static prompt and a dynamic response, this structure differs fundamentally from vision tasks. Consequently, a critical gap exists in the efficient acceleration of dLLMs.

3 Methodology

3.1 Preliminary

Inference Process of dLLMs. dLLMs generate text via a non-autoregressive, diffusion-based process that iteratively denoises a fully masked sequence into the target output. Our work focuses on accelerating this inference procedure. We use LLaDA as a representative example to illustrate it.

Let $\mathcal{T}$ be the token vocabulary and `[MASK]` $\in \mathcal{T}$ the special mask token. Given a prompt $\mathbf{c} = (c_1, \ldots, c_M)$, the model generates a response $\mathbf{y} = (y_1, \ldots, y_L)$ through K discrete denoising steps,

indexed by $k = K$ down to 0. Let $\mathbf{y}^{(k)} \in \mathcal{T}^L$ denote the intermediate state at step k, starting from a fully masked sequence:

$$\mathbf{y}^{(K)} = (\underbrace{\texttt{[MASK]}, \ldots, \texttt{[MASK]}}_{L \text{ times}}) \tag{1}$$

At each step k, a mask predictor f_ϕ estimates the distribution over the clean sequence:

$$P_\phi(\mathbf{y}|\mathbf{c}, \mathbf{y}^{(k)}) = f_\phi(\mathbf{c}, \mathbf{y}^{(k)}; \phi) \tag{2}$$

The most likely sequence $\hat{\mathbf{y}}^{(0)}$ is typically obtained via greedy decoding:

$$\hat{\mathbf{y}}^{(0)} = \arg\max_{\mathbf{y} \in \mathcal{T}^L} P_\phi(\mathbf{y}|\mathbf{c}, \mathbf{y}^{(k)}) \tag{3}$$

A transition function S then yields $\mathbf{y}^{(k-1)}$ by selectively updating tokens in $\mathbf{y}^{(k)}$ based on $\hat{\mathbf{y}}^{(0)}$:

$$\mathbf{y}^{(k-1)} = S(\hat{\mathbf{y}}^{(0)}, \mathbf{y}^{(k)}, \mathbf{c}, k) \tag{4}$$

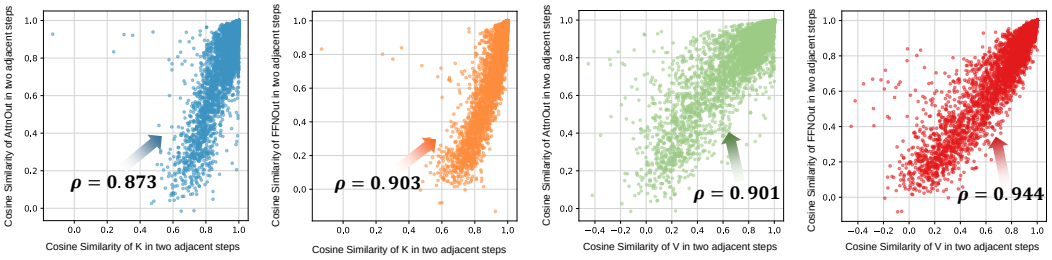
The specific strategy for S may involve confidence-based remasking or semi-autoregressive block updates. While this process enables flexible generation, it incurs high latency due to repeated recomputation across steps, particularly as K grows.

3.2 dLLM-Cache

To alleviate the inference inefficiency of dLLMs, we introduce **dLLM-Cache**, a training-free caching framework. The input prompt remains static across denoising steps, and its internal features are consistently stable, making it suitable for long-interval caching. In contrast, the response sequence evolves dynamically. However, this evolution is highly sparse, as only a small fraction of response tokens change significantly at each step. Such sparsity, evident in Figure 1, suggests that recomputing all response features in every step is often unnecessary.

To take advantage of this sparsity, dLLM-Cache selectively updates only a small fraction of response tokens that change most between adjacent steps. The challenge is to identify such tokens efficiently and accurately. Figure 2 reveals that the change in a response token's Value ($\mathbf{V}$) or Key ($\mathbf{K}$) vector, which is quantified by cosine similarity between current and cached versions, strongly correlates with changes in its subsequent Attention Output ($\mathbf{AttnOut}$) and Feedforward Network Output ($\mathbf{FFNOut}$). This strong correlation indicates that by monitoring the dynamics of earlier-stage features like $\mathbf{V}$, we can effectively identify tokens whose more complex downstream features are also likely to have significantly changed.

Based on this finding, we introduce our **V-verify** mechanism. It uses the cosine similarity between each response token's current $\mathbf{V}$ vector and its cached counterpart to identify tokens with the largest $\mathbf{V}$ changes. Only these selected tokens undergo a full feature recomputation and cache update.



(a) Correlation between **Key** and **AttnOut**

(b) Correlation between **Key** and **FFNOut**

(c) Correlation between **Value** and **AttnOut**

(d) Correlation between **Value** and **FFNOut**

Figure 2: **Correlation of response tokens' $\mathbf{K}$ or $\mathbf{V}$ changes with other feature changes**. We calculate the cosine similarity between the response tokens' $\mathbf{K}$ or $\mathbf{V}$ vectors and their cached counterparts at adjacent steps, select the 25% most dissimilar tokens, and compute the correlation between their similarity with (a) and (c) $\mathbf{AttnOut}$, or (b) and (d) $\mathbf{FFNOut}$ across adjacent steps.

Building on this core idea, the overall workflow of dLLM-Cache illustrated in Figure 3 is as follows: For each Transformer layer l, we store its $\mathbf{K}^{(l)}$, $\mathbf{V}^{(l)}$, $\mathbf{AttnOut}^{(l)}$, and $\mathbf{FFNOut}^{(l)}$ in a Prompt

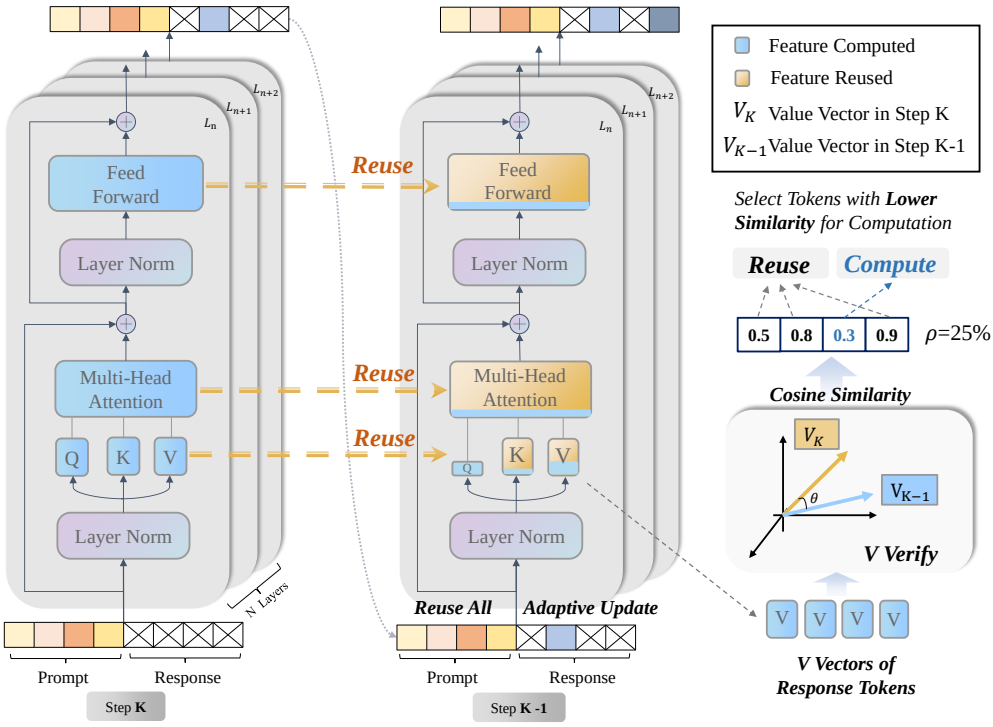


Figure 3: **The dLLM-Cache pipeline.** Prompt features are updated with long intervals, while response features are updated adaptively based on the similarity between newly computed and cached $\mathbf{V}$ vectors. Response features of tokens with low similarity are updated, and the rest are reused.

Cache $\mathcal{C}_p$ and a Response Cache $\mathcal{C}_r$, respectively. Caching is controlled by three hyperparameters: prompt refresh interval K_p, response refresh interval K_r, and adaptive update ratio $\rho \in [0, 1]$. The inference process generally involves:

Initialization At the beginning when $k = K$, compute all features from $(\mathbf{c}, \mathbf{y}^{(K)})$, storing prompt-related features in $\mathcal{C}_p$ and response-related features in $\mathcal{C}_r$.

Iterative Steps When $k = K-1$ to 1, for each step and layer:

- **Prompt:** If $k \equiv 0 \pmod{K_p}$, recompute and update $\mathcal{C}_p$; otherwise, reuse.
- **Response:** If $k \equiv 0 \pmod{K_r}$, fully recompute and update $\mathcal{C}_r$; otherwise, perform adaptive update detailed in Sec. 3.2.2.
- **Computation:** Use cached or updated features to execute layer l.

Termination Stop at $k = 0$ and return $\hat{\mathbf{y}}^{(0)}$.

The complete algorithmic workflow of dLLM-Cache is summarized in Algorithm 1.

3.2.1 Prompt Cache Management

Given that the input prompt $\mathbf{c}$ is static throughout the inference process, its corresponding features exhibit high temporal stability. dLLM-Cache leverages this by maintaining a Prompt Cache $\mathcal{C}_p$. At initialization where $k = K$, all prompt-related features $\mathbf{K}_p^{(l)}, \mathbf{V}_p^{(l)}, \mathbf{AttnOut}_p^{(l)}, \mathbf{FFNOut}_p^{(l)}$ for each layer l are computed and stored in $\mathcal{C}_p$. In subsequent denoising steps k:

- If $k \equiv 0 \pmod{K_p}$, the features in $\mathcal{C}_p$ are fully recomputed. Importantly, the calculation of $\mathbf{AttnOut}_p^{(l)}$ and subsequently $\mathbf{FFNOut}_p^{(l)}$ for the prompt tokens considers the current state of Key and Value features from the Response Cache $\mathcal{C}_r$ which might have been updated in the same step if $k \equiv 0 \pmod{K_r}$ or via adaptive updates. This ensures that prompt representations are conditioned on the evolving response.
- Otherwise, if $k \not\equiv 0 \pmod{K_p}$, all prompt-specific features are directly retrieved from $\mathcal{C}_p$.

This strategy significantly reduces the computational load associated with processing the static prompt, especially when K_p is large.

3.2.2 Response Cache with Adaptive Updates

Unlike prompts, response features $\mathbf{y}^{(k)}$ evolve across steps. However, most tokens change gradually, allowing selective updates. The response cache $\mathcal{C}_r$ is managed in two modes:

Full Refresh When $k \equiv 0 \pmod{K_r}$ or $k = K$, recompute all response features and update $\mathcal{C}_r$.

Adaptive Partial Update Otherwise:

- **Identify Change:** Compute cosine similarity s_j between current and cached Value vectors for each response token j:

$$s_j = \frac{(\mathbf{v}_{r,j}^{(l)})^\top \tilde{\mathbf{v}}_{r,j}^{(l)}}{\|\mathbf{v}_{r,j}^{(l)}\| \|\tilde{\mathbf{v}}_{r,j}^{(l)}\|} \tag{5}$$

- **Select Tokens:** Choose $\lfloor \rho L \rfloor$ tokens with lowest s_j to update, forming $\mathcal{I}_{\text{update}}$.
- **Recompute & Reuse:** Recompute $\mathbf{K}_r^{(l)}$, $\mathbf{V}_r^{(l)}$, $\mathbf{AttnOut}_r^{(l)}$, and $\mathbf{FFNOut}_r^{(l)}$ only for $j \in \mathcal{I}_{\text{update}}$; reuse cached values for others.
- **Update Cache:** Replace corresponding entries in $\mathcal{C}_r$ with updated values.

This adaptive update mechanism leverages temporal stability to minimize computation, while preserving generation quality through targeted feature refresh.

4 Experiments

4.1 Experiment Settings

Implementation Details We evaluated dLLM-Cache on two representative dLLMs: LLaDA 8B [Nie et al., 2025] and Dream 7B [Ye et al., 2025], each with Base and Instruct variants. Following the original inference configurations detailed in Appendix A, we conducted our experiments across various benchmarks. For all models, we applied a fixed adaptive update ratio of $\rho = 0.25$. The prompt refresh interval K_p and response refresh interval K_r are specified in the corresponding result tables. All experiments were conducted on the NVIDIA RTX 4090 and H100 80GB SMX GPUs.

Evaluation Metrics We evaluate the acceleration and model quality preservation of dLLM-Cache using several metrics. Throughput is measured as Tokens Per Second (TPS), reflecting inference speed. Computational cost is calculated as the average Floating Point Operations (FLOPs) per token. Task performance is assessed using benchmark scores, like accuracy on GSM8K [Cobbe et al., 2021], ensuring dLLM-Cache achieves efficiency gains without compromising model performance. The testing of TPS and FLOPs was performed on a single RTX 4090 GPU.

4.2 Main Results

The main results comparing baseline model inference with and without dLLM-Cache are presented in Table 1 for LLaDA 8B and Table 2 for Dream 7B, respectively. These results consistently demonstrate that dLLM-Cache brings significant improvements in inference efficiency while achieving lossless acceleration in most cases. For the LLaDA 8B model, dLLM-Cache demonstrates exceptional gains. Notably, on the GPQA benchmark, when applied to the LLaDA Instruct model, dLLM-Cache delivered up to an 8.08× speedup, reducing FLOPs from 22.07T to 2.73T losslessly. Similarly, when applied to the Dream 7B model, dLLM-Cache also achieved surprising efficiency enhancements. On the GSM8K benchmark, the Dream Base model with dLLM-Cache achieved a 6.90× speedup while maintaining accuracy without any loss. In practical applications, the values of K_p and K_r can be adjusted to achieve the desired emphasis between acceleration and quality as needed.

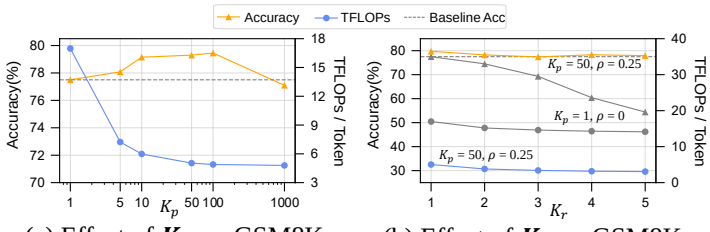
4.3 Ablation Study

Effect of Cache Refresh Interval K_p and K_r. We analyzed how refresh intervals affect efficiency and accuracy. As shown in Figure 4(a), increasing the prompt interval K_p substantially reduces

Table 1: **Comparison of LLaDA 8B with and without dLLM-Cache** on 8 benchmarks.

Task	Method	Inference Efficiency				Performance
		TPS↑	Speed(TPS)↑	FLOPs(T)↓	Speed(FLOPs)↑	Score↑
Mathematics & Science						
GSM8K	LLaDA Base	7.32	1.00×	16.12	1.00×	69.06
	+ Cache (K_p = 100, K_r = 6)	31.43$_{+24.11}$	4.29×$_{+3.29}$	2.76$_{-13.36}$	5.84×$_{+4.84}$	67.32$_{-1.74}$
	+ Cache (K_p = 25, K_r = 5)	23.19$_{+15.87}$	3.17×$_{+2.17}$	3.21$_{-12.91}$	5.02×$_{+4.02}$	70.66$_{+1.60}$
	LLaDA Instruct	6.95	1.00×	16.97	1.00×	77.48
	+ Cache (K_p = 50, K_r = 7)	29.75$_{+22.80}$	4.28×$_{+3.28}$	2.92$_{-14.05}$	5.81×$_{+4.81}$	78.54$_{+1.06}$
GPQA	LLaDA Base	5.12	1.00×	22.97	1.00×	31.91
	+ Cache (K_p = 100, K_r = 8)	25.23$_{+20.11}$	4.93×$_{+3.93}$	3.20$_{-19.77}$	7.18×$_{+6.18}$	32.81$_{+0.90}$
	LLaDA Instruct	5.33	1.00×	22.07	1.00×	29.01
	+ Cache (K_p = 50, K_r = 6)	28.01$_{+22.68}$	5.26×$_{+4.26}$	2.73$_{-19.34}$	8.08×$_{+7.08}$	29.01$_{+0.00}$
Math	LLaDA Base	8.31	1.00×	14.11	1.00×	30.84
	+ Cache (K_p = 50, K_r = 8)	33.92$_{+25.61}$	4.08×$_{+3.08}$	2.61$_{-11.50}$	5.41×$_{+4.41}$	29.84$_{-1.00}$
	LLaDA Instruct	23.65	1.00×	5.16	1.00×	22.32
	+ Cache (K_p = 50, K_r = 1)	31.02$_{+7.37}$	1.31×$_{+0.31}$	3.96$_{-1.20}$	1.30×$_{+0.30}$	22.52$_{+0.20}$
General Tasks						
MMLU-pro	LLaDA Base	14.08	1.00×	8.40	1.00×	24.26
	+ Cache (K_p = 100, K_r = 6)	45.75$_{+31.67}$	3.25×$_{+2.25}$	2.15$_{-6.25}$	3.91×$_{+2.91}$	24.69$_{+0.43}$
	LLaDA Instruct	14.01	1.00×	8.46	1.00×	36.41
	+ Cache (K_p = 51, K_r = 3)	39.63$_{+25.62}$	2.83×$_{+1.83}$	2.62$_{-5.84}$	3.23×$_{+2.23}$	36.08$_{-0.33}$
MMLU	LLaDA Base	8.09	1.00×	14.56	1.00×	63.99
	+ Cache (K_p = 100, K_r = 6)	33.52$_{+25.43}$	4.14×$_{+3.14}$	2.64$_{-11.92}$	5.52×$_{+4.52}$	64.26$_{+0.27}$
	LLaDA Instruct	10.12	1.00×	11.85	1.00×	61.24
	+ Cache (K_p = 100, K_r = 7)	21.23$_{+11.11}$	2.10×$_{+1.10}$	4.50$_{-7.35}$	2.63×$_{+1.63}$	62.82$_{+1.58}$
BBH	LLaDA Base	6.41	1.00×	18.29	1.00×	44.77
	+ Cache (K_p = 50, K_r = 6)	27.90$_{+21.49}$	4.35×$_{+3.35}$	3.09$_{-15.20}$	5.92×$_{+4.92}$	45.04$_{+0.27}$
	LLaDA Instruct	6.18	1.00×	18.98	1.00×	51.49
	+ Cache (K_p = 100, K_r = 5)	27.55$_{+21.37}$	4.46×$_{+3.46}$	3.08$_{-15.90}$	6.16×$_{+5.16}$	51.98$_{+0.49}$
Code						
MBPP	LLaDA Base	7.87	1.00×	14.91	1.00×	40.80
	+ Cache (K_p = 50, K_r = 4)	30.74$_{+22.87}$	3.91×$_{+2.91}$	2.99$_{-11.92}$	4.99×$_{+3.99}$	39.00$_{-1.80}$
	+ Cache (K_p = 25, K_r = 4)	24.61$_{+16.74}$	3.13×$_{+2.13}$	3.07$_{-11.84}$	4.86×$_{+3.86}$	40.60$_{-0.20}$
	LLaDA Instruct	7.55	1.00×	15.53	1.00×	39.20
	+ Cache (K_p = 100, K_r = 5)	31.73$_{+24.18}$	4.20×$_{+3.20}$	2.80$_{-12.73}$	5.55×$_{+4.55}$	39.60$_{+0.40}$
HumanEval	LLaDA Base	19.98	1.00×	6.03	1.00×	32.92
	+ Cache (K_p = 100, K_r = 5)	51.96$_{+31.98}$	2.60×$_{+1.60}$	2.04$_{-3.99}$	2.96×$_{+1.96}$	32.31$_{-0.61}$
	+ Cache (K_p = 50, K_r = 5)	30.68$_{+10.70}$	1.54×$_{+0.54}$	2.07$_{-3.96}$	2.91×$_{+1.91}$	32.92$_{+0.00}$
	LLaDA Instruct	10.57	1.00×	11.10	1.00×	38.71
	+ Cache (K_p = 50, K_r = 8)	47.29$_{+36.72}$	4.47×$_{+3.47}$	1.83$_{-9.27}$	6.07×$_{+5.07}$	37.19$_{-1.52}$
	+ Cache (K_p = 25, K_r = 5)	44.77$_{+34.20}$	4.24×$_{+3.24}$	2.05$_{-9.05}$	5.41×$_{+4.41}$	39.02$_{+0.31}$

FLOPs without hurting accuracy, confirming that infrequent prompt updates suffice. Figure 4(b) highlights the importance of response updates. Without prompt caching or adaptive updates ($K_p = 1$, $\rho = 0$, gray line), accuracy drops sharply. In contrast, our setting ($K_p = 50$, $\rho = 0.25$, orange and blue line) maintains high accuracy with much lower cost. This validates our strategy of combining long prompt intervals with short response intervals and adaptive updates.



(a) Effect of K_p on GSM8K (b) Effect of K_r on GSM8K

Figure 4: **Effect of cache refresh intervals** using LLaDA 8B Instruct. (a) Varying K_p with $K_r = 1$, $\rho = 0$. (b) Varying K_r under two settings: baseline with $K_p = 1$, $\rho = 0$ in gary and our setup $K_p = 50$, $\rho = 0.25$ in Table 1.

Table 2: **Comparison of Dream 7B with and without dLLM-Cache** on 8 benchmarks.

Task	Configuration	Inference Efficiency				Performance
		TPS↑	Speed(TPS)↑	FLOPs(T)↓	Speed(FLOPs)↑	Score↑
Mathematics & Science						
GSM8K	Dream Base	6.36	1.00×	19.59	1.00×	76.95
	+ Cache (K_p = 100, K_r = 8)	32.44$_{+26.08}$	5.10×$_{+4.10}$	2.84$_{-16.75}$	6.90×$_{+5.90}$	76.95$_{+0.00}$
	Dream Instruct	6.39	1.00×	19.57	1.00×	77.55
	+ Cache (K_p = 25, K_r = 2)	24.52$_{+18.13}$	3.84×$_{+2.84}$	4.24$_{-15.33}$	4.62×$_{+3.61}$	76.80$_{-0.75}$
GPQA	Dream Base	5.80	1.00×	21.66	1.00×	33.92
	+ Cache (K_p = 100, K_r = 8)	30.95$_{+25.15}$	5.33×$_{+4.33}$	3.03$_{-18.63}$	7.15×$_{+6.15}$	34.15$_{+0.23}$
	Dream Instruct	5.53	1.00×	22.63	1.00×	34.38
	+ Cache (K_p = 10, K_r = 8)	21.98$_{+16.45}$	3.97×$_{+2.97}$	4.69$_{-17.94}$	4.83×$_{+3.82}$	33.93$_{-0.45}$
Math	Dream Base	9.40	1.00×	13.31	1.00×	38.68
	+ Cache (K_p = 100, K_r = 4)	36.89$_{+27.49}$	3.92×$_{+2.92}$	2.61$_{-10.70}$	5.10×$_{+4.10}$	37.94$_{-0.74}$
	Dream Instruct	8.85	1.00×	14.11	1.00×	38.28
	+ Cache (K_p = 50, K_r = 1)	23.52$_{+14.67}$	2.66×$_{+1.66}$	4.66$_{-9.45}$	3.03×$_{+2.03}$	37.62$_{-0.66}$
General Tasks						
MMLU-pro	Dream Base	15.61	1.00×	7.92	1.00×	24.13
	+ Cache (K_p = 25, K_r = 2)	35.86$_{+20.25}$	2.30×$_{+1.30}$	2.89$_{-5.03}$	2.74×$_{+1.74}$	23.86$_{-0.27}$
	Dream Instruct	15.40	1.00×	7.98	1.00×	43.79
	+ Cache (K_p = 5, K_r = 1)	23.98$_{+8.58}$	1.56×$_{+0.56}$	4.77$_{-3.21}$	1.67×$_{+0.67}$	43.96$_{+0.17}$
MMLU	Dream Base	9.10	1.00×	13.73	1.00×	73.49
	+ Cache (K_p = 100, K_r = 2)	31.07$_{+21.97}$	3.41×$_{+2.41}$	3.27$_{-10.46}$	4.20×$_{+3.20}$	73.20$_{-0.29}$
	Dream Instruct	8.45	1.00×	14.75	1.00×	73.40
	+ Cache (K_p = 100, K_r = 8)	38.01$_{+29.56}$	4.50×$_{+3.50}$	2.42$_{-12.33}$	6.10×$_{+5.10}$	73.42$_{+0.02}$
BBH	Dream Base	7.24	1.00×	17.25	1.00×	52.25
	+ Cache (K_p = 25, K_r = 4)	29.61$_{+22.37}$	4.09×$_{+3.09}$	3.35$_{-13.90}$	5.15×$_{+4.15}$	51.66$_{-0.59}$
	Dream Instruct	6.98	1.00×	17.90	1.00×	57.07
	+ Cache (K_p = 10, K_r = 2)	22.31$_{+15.33}$	3.20×$_{+2.20}$	4.82$_{-13.08}$	3.71×$_{+2.71}$	57.07$_{+0.00}$
Code						
MBPP	Dream Base	8.91	1.00×	14.06	1.00×	54.20
	+ Cache (K_p = 25, K_r = 8)	35.69$_{+26.78}$	4.01×$_{+3.01}$	2.66$_{-11.40}$	5.29×$_{+4.29}$	54.20$_{+0.00}$
	Dream Instruct	8.46	1.00×	14.65	1.00×	57.00
	+ Cache (K_p = 10, K_r = 8)	29.77$_{+21.31}$	3.52×$_{+2.52}$	3.33$_{-11.32}$	4.40×$_{+3.40}$	56.80$_{-0.20}$
HumanEval	Dream Base	21.43	1.00×	5.68	1.00×	58.53
	+ Cache (K_p = 5, K_r = 1)	27.40$_{+5.97}$	1.28×$_{+0.28}$	4.17$_{-1.51}$	1.36×$_{+0.36}$	57.31$_{-1.22}$
	Dream Instruct	17.88	1.00×	6.84	1.00×	57.92
	+ Cache (K_p = 50, K_r = 1)	28.03$_{+10.15}$	1.57×$_{+0.57}$	3.94$_{-2.90}$	1.74×$_{+0.74}$	56.09$_{-1.83}$

Table 3: **Comparison of LLaDA 8B Base with other representative LLM** on GSM8K. The superscripts * and † denote LLaDA with 256 and 128 decoding steps, respectively.

Method	TPS↑	Speed(TPS) ↑	Accuracy↑	GPU Memory (GB)↓
LLaMA3 8B [Dubey et al., 2024]	47.73	1.00×	49.05	16.06
LLaDA*	7.37$_{-40.36}$	1.00×	69.06$_{+20.01}$	16.94
+ Cache* (K_p = 100, K_r = 6)	25.02$_{-22.71}$	3.39×	67.32$_{+18.27}$	17.93
LLaDA†	14.77$_{-32.96}$	1.00×	64.14$_{+15.09}$	16.94
+ Cache† (K_p = 50, K_r = 8)	49.15$_{+1.42}$	3.33×	62.32$_{+13.27}$	17.93

Effect of Update Ratio ρ and Selection Strategy. We investigated how different token selection strategies impact performance under varying adaptive update ratios ρ. Figure 5 reports accuracy and computational cost on GSM8K when using three strategies: **V-verify**, **K-verify**, and random selection. Both similarity-based strategies consistently outperform random selection across a wide range of ρ values, confirming the importance of dynamic, feature-driven updates. In particular, value-based selection achieves the highest accuracy around $\rho = 0.25$, while requiring significantly fewer FLOPs than full recomputation. This suggests that moderate, targeted updates, *e.g.*, $\rho \approx 0.25$ strike a favorable trade-off between efficiency and output quality.

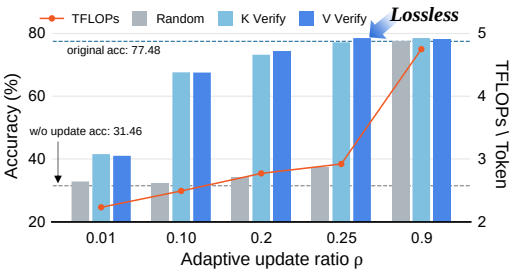


Figure 5: **Effect of token selection strategy** on GSM8K using LLaDA 8B Instruct model under varying update ratios ρ.

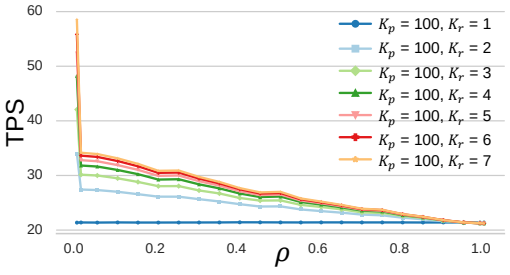


Figure 6: **TPS versus ρ.** A notable decrease in TPS persists at minimal ρ values, highlighting fixed operational overheads associated with initiating any selective update.

Impact of Similarity Metric. We compared cosine similarity and L2 distance as similarity metrics for **V-verify**. On GSM8K with LLaDA 8B Instruct, cosine similarity achieved 78.54% accuracy, significantly outperforming L2 distance at 55.95%. This shows that cosine similarity better captures semantic change, and we adopt it as the default throughout our method.

5 Discussion

Effectiveness on Long-Prompt Scenarios The benefits of dLLM-Cache are particularly pronounced in scenarios involving long input prompts, common in tasks like document-based question answering. Our Long-Interval Prompt Caching mechanism significantly curtails redundant computations for the extensive static prompt portion by refreshing its cache only at long intervals. For instance, when applying dLLM-Cache to the LLaDA 8B Base model on the LongBench-HotpotQA [Bai et al., 2023] task, we not only achieved a **9.1× speedup** over the unaccelerated baseline but also observed a performance improvement, with the F1 score increasing from 34.56 to 36.10. This highlights the particular suitability of dLLM-Cache for dLLM applications requiring extensive contextual understanding, where our caching strategy for long static prompts can be maximally leveraged.

Storage Overhead of Caching. dLLM-Cache stores four types of intermediate features per layer, including $\mathbf{K}$, $\mathbf{V}$, $\mathbf{AttnOut}$, and $\mathbf{FFNOut}$. Theoretically, the total cache size scales with the number of tokens T, embedding dimension d, and number of layers L. Specifically, the storage cost is $T \times d \times 4 \times L$, as proven in Appendix B. Since only one version per layer is cached, the overall memory footprint remains stable. As shown in Table 3, on GSM8K with LLaDA 8B Base, the peak GPU memory usage is 16.94GB without caching, 17.93GB with dLLM-Cache, and 16.06GB for LLaMA3 8B (autoregressive with KV cache). Our method introduces moderate overhead but achieves much higher accuracy compared to standard AR models, *e.g.*, LLaMA3 8B.

Cost of V-verify and the Fixed Update Overheads. Our **V-verify** mechanism uses lightweight $\mathbf{V}$ vector similarity for identifying dynamic tokens. While **V-verify** itself is computationally inexpensive, as illustrated in Figure 6, practical speedup from adaptive partial updates is constrained by fixed operational overheads. Figure 6 shows a notable decrease in TPS as the update ratio ρ approaches zero. This base cost arises because initiating any selective recomputation ($\rho > 0$) triggers non-negligible system-level latencies, *e.g.*, for GPU kernel management and data movement that are not strictly proportional to the number of updated tokens. Consequently, at very low ρ values, these fixed overheads dominate, limiting further run time savings. An optimal ρ must balance these fixed costs against saved dynamic computation, while preserving model quality. Figure 5 suggests $\rho \approx 0.25$ offers an effective trade-off between the costs of activating selective updates and the benefits of reduced computation, optimizing overall efficiency and fidelity.

6 Conclusion

We present dLLM-Cache, a training-free and model-agnostic caching method for accelerating inference in diffusion-based large language models. Experiments on LLaDA and Dream show that dLLM-Cache achieves significant speedup without compromising generation quality.

A current limitation is that existing open-source dLLMs are restricted to 8B parameters, preventing evaluation on larger models such as 33B or 70B. We believe that dLLM-Cache will deliver even greater acceleration benefits at larger scales, and we leave this for future work.

References

Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. Structured denoising diffusion models in discrete state-spaces. *Advances in Neural Information Processing Systems*, 34:17981–17993, 2021.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

Lukas Berglund, Meg Tong, Max Kaufmann, Mikita Balesni, Asa Cooper Stickland, Tomasz Korbak, and Owain Evans. The reversal curse: Llms trained on" a is b" fail to learn" b is a". *arXiv preprint arXiv:2309.12288*, 2023.

Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

Senmao Li, Taihang Hu, Fahad Shahbaz Khan, Linxuan Li, Shiqi Yang, Yaxing Wang, Ming-Ming Cheng, and Jian Yang. Faster diffusion: Rethinking the role of unet encoder in diffusion models. *arXiv preprint arXiv:2312.09608*, 2023.

Jiacheng Liu, Chang Zou, Yuanhuiyi Lyu, Junjie Chen, and Linfeng Zhang. From reusing to forecasting: Accelerating diffusion models with taylorseers. *arXiv preprint arXiv:2503.06923*, 2025.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion language modeling by estimating the ratios of the data distribution. *arXiv preprint arXiv:2310.16834*, 2023.

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

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models, 2025. URL `https://arxiv.org/abs/2502.09992`.

OpenAI. ChatGPT: Optimizing Language Models for Dialogue. *OpenAI blog*, November 2022. URL `https://openai.com/blog/chatgpt/`.

Jingyang Ou, Shen Nie, Kaiwen Xue, Fengqi Zhu, Jiacheng Sun, Zhenguo Li, and Chongxuan Li. Your absorbing discrete diffusion secretly models the conditional distributions of clean data. *arXiv preprint arXiv:2406.03736*, 2024.

William Peebles and Saining Xie. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4195–4205, 2023.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5:606–624, 2023.

Alec Radford. Improving language understanding by generative pre-training, 2018.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

Pratheba Selvaraju, Tianyu Ding, Tianyi Chen, Ilya Zharkov, and Luming Liang. Fora: Fast-forward caching in diffusion transformer acceleration. *arXiv preprint arXiv:2407.01425*, 2024.

Jiaxin Shi, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis K Titsias. Simplified and generalized masked diffusion for discrete data. *arXiv preprint arXiv:2406.04329*, 2024.

Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International conference on machine learning*, pages 2256–2265. PMLR, 2015.

Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. In *International Conference on Learning Representations*, 2021.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7b, 2025. URL `https://hkunlp.github.io/blog/2025/dream`.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

Chang Zou, Xuyang Liu, Ting Liu, Siteng Huang, and Linfeng Zhang. Accelerating diffusion transformers with token-wise feature caching. *arXiv preprint arXiv:2410.05317*, 2024a.

Chang Zou, Evelyn Zhang, Runlin Guo, Haohang Xu, Conghui He, Xuming Hu, and Linfeng Zhang. Accelerating diffusion transformers with dual feature caching. *arXiv preprint arXiv:2412.18911*, 2024b.

A Experimental Details

This section provides the detailed configuration settings used in our experiments across a variety of tasks for both the Instruct and Base variants of the evaluated diffusion-based large language models. For each task, we report the number of denoising steps, the block length, the total generation length, the remasking strategy, and the number of few-shot examples used (if any). All models use the low-confidence remasking strategy unless otherwise specified.

Table 4: Experimental settings for Instruct model across selected benchmarks.

Task	Steps	Block Len	Gen Len	Few-shot
MMLU	3	3	3	5
MMLU-pro	256	256	256	0
Hellaswag	3	3	3	0
ARC-C	512	512	512	0
GSM8K	256	8	256	4
Math	256	256	256	0
GPQA	128	64	128	5
HumanEval	512	32	512	0
MBPP	512	32	512	3

B Proof of Storage Overhead of Caching

Theorem: The storage overhead of caching in our method is $O(T \times d \times 4 \times L)$, where T is the number of tokens, d is the embedding dimension, and L is the number of layers.

Proof. We first define the memory required for each layer of the model. In our method, four types of intermediate features are stored per layer: $\mathbf{K}$, $\mathbf{V}$, $\mathbf{AttnOut}$, and $\mathbf{FFNOut}$. Each feature has a size of $T \times d$, where T is the number of tokens and d is the embedding dimension.

Let M_{layer} denote the memory required for each layer. Since four feature types are cached per layer, the memory required for one layer is:

$$M_{\text{layer}} = 4 \times T \times d$$

This accounts for the four different feature types stored per token in the layer.

Now, consider the entire model, which consists of L layers. The total memory required for caching all layers is simply the memory required for one layer multiplied by the number of layers:

$$M_{\text{total}} = L \times M_{\text{layer}} = L \times 4 \times T \times d$$

Next, we consider the precision used to store these features. In our method, we use bfloat16 precision, where each element requires 2 bytes of memory. Therefore, the total memory required for storing all features in terms of bytes is:

$$M_{\text{total}} = 2 \times L \times 4 \times T \times d \text{ bytes}$$

Finally, in asymptotic analysis, we focus on the growth rate of the memory overhead and ignore constant factors such as the factor of 2 bytes for precision. Therefore, the storage overhead grows as:

$$O(T \times d \times 4 \times L)$$

This completes the proof. □

C Core Algorithmic Workflow of dLLM-Cache

Algorithm 1 outlines the full forward computation process of dLLM-Cache, our training-free adaptive caching framework for diffusion-based large language models. At each denoising step, the algorithm dynamically determines whether to refresh prompt and/or response features based on predefined cache intervals (K_p for prompt, K_r for response). When neither full refresh condition is met, dLLM-Cache employs an adaptive update mechanism that selectively recomputes features for response tokens exhibiting the most significant semantic drift, as measured by value vector similarity. This selective caching strategy enables substantial computational savings without compromising generation quality, and is compatible with arbitrary Transformer-based denoising networks.

Algorithm 1 dLLM-Cache: Main Inference Algorithm

Require: Prompt $\mathbf{c}$, initial masked sequence $\mathbf{y}^{(K)}$, denoising steps K, prompt refresh interval K_p, response refresh interval K_r, adaptive update ratio ρ
Ensure: Final prediction $\hat{\mathbf{y}}^{(0)}$

/* Initialize caches at step $k = K$ */
$\mathcal{C}_p, \mathcal{C}_r \leftarrow$ InitializeCache($\mathbf{c}, \mathbf{y}^{(K)}$) ▷ Algorithm 2
Generate prediction $\hat{\mathbf{y}}^{(0)}$ using model f_ϕ ▷ Needs initial pass or separate handling
$\mathbf{y}^{(K-1)} \leftarrow S(\hat{\mathbf{y}}^{(0)}, \mathbf{y}^{(K)}, \mathbf{c}, K)$
for $k = K - 1$ **down to** 1 **do**
    $\mathbf{x}_{layer\_in} \leftarrow [\mathbf{c}; \mathbf{y}^{(k)}]$ ▷ Initial input for layer 1 at step k
    for each layer l in the Transformer network **do**
        /* Determine refresh conditions based on intervals */
        refresh_prompt $\leftarrow (k \bmod K_p = 0)$ ▷ Refresh prompt every K_p steps
        refresh_response $\leftarrow (k \bmod K_r = 0)$ ▷ Refresh response every K_r steps
        /* Cache usage strategy based on refresh conditions */
        if refresh_prompt **and** refresh_response **then**
            $\mathbf{x}_{layer\_out}, \mathcal{C}_p, \mathcal{C}_r \leftarrow$ FullRefresh($\mathbf{x}_{layer\_in}, l, \mathcal{C}_p, \mathcal{C}_r$) ▷ Algorithm 3
        else if refresh_prompt **and not** refresh_response **then**
            $\mathbf{x}_{layer\_out}, \mathcal{C}_p, \mathcal{C}_r \leftarrow$ RefreshPromptOnly($\mathbf{x}_{layer\_in}, l, \mathcal{C}_p, \mathcal{C}_r$) ▷ Algorithm 4
        else if not refresh_prompt **and** refresh_response **then**
            $\mathbf{x}_{layer\_out}, \mathcal{C}_p, \mathcal{C}_r \leftarrow$ RefreshResponseOnly($\mathbf{x}_{layer\_in}, l, \mathcal{C}_p, \mathcal{C}_r$) ▷ Algorithm 5
        else
            /* When neither needs full refresh */
            $\mathbf{x}_{layer\_out}, \mathcal{C}_p, \mathcal{C}_r \leftarrow$ AdaptiveUpdate($\mathbf{x}_{layer\_in}, l, \mathcal{C}_p, \mathcal{C}_r, \rho$) ▷ Algorithm 6
        end if
        $\mathbf{x}_{layer\_in} \leftarrow \mathbf{x}_{layer\_out}$ ▷ Update input for the next layer
    end for ▷ End layer loop
    Generate prediction $\hat{\mathbf{y}}^{(0)}$ using model f_ϕ with final layer output $\mathbf{x}_{layer\_out}$
    $\mathbf{y}^{(k-1)} \leftarrow S(\hat{\mathbf{y}}^{(0)}, \mathbf{y}^{(k)}, \mathbf{c}, k)$ ▷ Apply transition function
end for ▷ End step loop
return final prediction $\hat{\mathbf{y}}^{(0)}$

Algorithm 2 dLLM-Cache: Cache Structure and Initialization

Require: Prompt $\mathbf{c}$, initial masked sequence $\mathbf{y}^{(K)}$, Transformer network with L layers

/* Cache Structure Definition */
for layer $l \in \{1, 2, \ldots, L\}$ **do**
    $\mathcal{C}_p[l][\texttt{kv\_cache}] \leftarrow \{\}$ ▷ Prompt key-value cache
    $\mathcal{C}_p[l][\texttt{attn}] \leftarrow \{\}$ ▷ Prompt attention output cache
    $\mathcal{C}_p[l][\texttt{mlp}] \leftarrow \{\}$ ▷ Prompt FFN output cache
    $\mathcal{C}_r[l][\texttt{kv\_cache}] \leftarrow \{\}$ ▷ Response key-value cache
    $\mathcal{C}_r[l][\texttt{attn}] \leftarrow \{\}$ ▷ Response attention output cache
    $\mathcal{C}_r[l][\texttt{mlp}] \leftarrow \{\}$ ▷ Response FFN output cache
end for
/* Initial Caching (Step $k = K$) */
$\mathbf{x}_{in} \leftarrow [\mathbf{c}; \mathbf{y}^{(K)}]$ ▷ Concatenated input for the first layer
for layer $l \in \{1, 2, \ldots, L\}$ **do**
    /* -- Attention Block -- */
    $\mathbf{x}_{norm} \leftarrow \text{LayerNorm}(\mathbf{x}_{in})$
    $\mathbf{Q}, \mathbf{K}, \mathbf{V} \leftarrow \text{Q\_proj}(\mathbf{x}_{norm}), \text{K\_proj}(\mathbf{x}_{norm}), \text{V\_proj}(\mathbf{x}_{norm})$
    /* Split K, V for caching */
    $\mathbf{K}_p, \mathbf{K}_r \leftarrow \mathbf{K}_{1:|\mathbf{c}|}, \mathbf{K}_{|\mathbf{c}|+1:}$
    $\mathbf{V}_p, \mathbf{V}_r \leftarrow \mathbf{V}_{1:|\mathbf{c}|}, \mathbf{V}_{|\mathbf{c}|+1:}$
    $\mathcal{C}_p[l][\texttt{kv\_cache}] \leftarrow \{\mathbf{K}_p, \mathbf{V}_p\}$ ▷ Store prompt KV
    $\mathcal{C}_r[l][\texttt{kv\_cache}] \leftarrow \{\mathbf{K}_r, \mathbf{V}_r\}$ ▷ Store response KV
    $\mathbf{AttnOut} \leftarrow \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$ ▷ Compute combined attention
    /* Split AttnOut for caching */
    $\mathbf{AttnOut}_p, \mathbf{AttnOut}_r \leftarrow \mathbf{AttnOut}_{1:|\mathbf{c}|}, \mathbf{AttnOut}_{|\mathbf{c}|+1:}$
    $\mathcal{C}_p[l][\texttt{attn}] \leftarrow \mathbf{AttnOut}_p$ ▷ Store prompt attention output
    $\mathcal{C}_r[l][\texttt{attn}] \leftarrow \mathbf{AttnOut}_r$ ▷ Store response attention output
    $\mathbf{h} \leftarrow \mathbf{x}_{in} + \mathbf{AttnOut}$ ▷ Post-attention residual
    /* -- FFN Block -- */
    $\mathbf{h}_{norm} \leftarrow \text{LayerNorm}(\mathbf{h})$
    $\mathbf{FFNOut} \leftarrow \text{FFN}(\mathbf{h}_{norm})$ ▷ Compute combined FFN output
    /* Split FFNOut for caching */
    $\mathbf{FFNOut}_p, \mathbf{FFNOut}_r \leftarrow \mathbf{FFNOut}_{1:|\mathbf{c}|}, \mathbf{FFNOut}_{|\mathbf{c}|+1:}$
    $\mathcal{C}_p[l][\texttt{mlp}] \leftarrow \mathbf{FFNOut}_p$ ▷ Store prompt FFN output
    $\mathcal{C}_r[l][\texttt{mlp}] \leftarrow \mathbf{FFNOut}_r$ ▷ Store response FFN output
    $\mathbf{x}_{out} \leftarrow \mathbf{h} + \mathbf{FFNOut}$ ▷ Final residual. Note: Code uses dropout here.
    $\mathbf{x}_{in} \leftarrow \mathbf{x}_{out}$ ▷ Update input for the next layer
end for
return $\mathcal{C}_p, \mathcal{C}_r$ ▷ Initialized caches

Algorithm 3 dLLM-Cache: Case 1 - Full Refresh

Require: Layer input $\mathbf{x}_{in}$, layer index l, caches $\mathcal{C}_p$ and $\mathcal{C}_r$

▷ $\mathbf{x}_{in}$ is the output of layer $l-1$, or $[\mathbf{c}; \mathbf{y}^{(k)}]$ for $l = 1$

`/* Case 1: Refresh both prompt and response */`
`/* -- Attention Block -- */`
$\mathbf{x}_{norm} \leftarrow \text{LayerNorm}(\mathbf{x}_{in})$
$\mathbf{Q}, \mathbf{K}, \mathbf{V} \leftarrow \text{Q\_proj}(\mathbf{x}_{norm}), \text{K\_proj}(\mathbf{x}_{norm}), \text{V\_proj}(\mathbf{x}_{norm})$
`/* Split K, V for caching */`
$\mathbf{K}_p, \mathbf{K}_r \leftarrow \mathbf{K}_{1:|\mathbf{c}|}, \mathbf{K}_{|\mathbf{c}|+1:}$
$\mathbf{V}_p, \mathbf{V}_r \leftarrow \mathbf{V}_{1:|\mathbf{c}|}, \mathbf{V}_{|\mathbf{c}|+1:}$
$\mathcal{C}_p[l][\texttt{kv\_cache}] \leftarrow \{\mathbf{K}_p, \mathbf{V}_p\}$ ▷ Update prompt KV cache
$\mathcal{C}_r[l][\texttt{kv\_cache}] \leftarrow \{\mathbf{K}_r, \mathbf{V}_r\}$ ▷ Update response KV cache
$\mathbf{AttnOut} \leftarrow \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$ ▷ Compute combined attention
`/* Split AttnOut for caching */`
$\mathbf{AttnOut}_p, \mathbf{AttnOut}_r \leftarrow \mathbf{AttnOut}_{1:|\mathbf{c}|}, \mathbf{AttnOut}_{|\mathbf{c}|+1:}$
$\mathcal{C}_p[l][\texttt{attn}] \leftarrow \mathbf{AttnOut}_p$ ▷ Update prompt attention cache
$\mathcal{C}_r[l][\texttt{attn}] \leftarrow \mathbf{AttnOut}_r$ ▷ Update response attention cache
$\mathbf{h} \leftarrow \mathbf{x}_{in} + \mathbf{AttnOut}$ ▷ Post-attention residual
`/* -- FFN Block -- */`
$\mathbf{h}_{norm} \leftarrow \text{LayerNorm}(\mathbf{h})$
$\mathbf{FFNOut} \leftarrow \text{FFN}(\mathbf{h}_{norm})$ ▷ Compute combined FFN output
`/* Split FFNOut for caching */`
$\mathbf{FFNOut}_p, \mathbf{FFNOut}_r \leftarrow \mathbf{FFNOut}_{1:|\mathbf{c}|}, \mathbf{FFNOut}_{|\mathbf{c}|+1:}$
$\mathcal{C}_p[l][\texttt{mlp}] \leftarrow \mathbf{FFNOut}_p$ ▷ Update prompt FFN cache
$\mathcal{C}_r[l][\texttt{mlp}] \leftarrow \mathbf{FFNOut}_r$ ▷ Update response FFN cache
$\mathbf{x}_{out} \leftarrow \mathbf{h} + \mathbf{FFNOut}$ ▷ Final residual.
return $\mathbf{x}_{out}, \mathcal{C}_p, \mathcal{C}_r$ ▷ Return layer output and updated caches

Algorithm 4 dLLM-Cache: Case 2 - Refresh Prompt Only

Require: Layer input $\mathbf{x}_{in}$, layer index l, caches $\mathcal{C}_p$ and $\mathcal{C}_r$

▷ $\mathbf{x}_{in}$ is the output of layer $l-1$

`/* Case 2: Refresh prompt only, reuse response features */`
$\mathbf{x}_{p\_in} \leftarrow \mathbf{x}_{in,1:|\mathbf{c}|}$ ▷ Layer's prompt input part
`/* Compute fresh prompt features */`
$\mathbf{x}_{p\_norm} \leftarrow \text{LayerNorm}(\mathbf{x}_{p\_in})$
$\mathbf{Q}_p \leftarrow \text{Q\_proj}(\mathbf{x}_{p\_norm})$; $\mathbf{K}_p \leftarrow \text{K\_proj}(\mathbf{x}_{p\_norm})$; $\mathbf{V}_p \leftarrow \text{V\_proj}(\mathbf{x}_{p\_norm})$
$\mathcal{C}_p[l][\texttt{kv\_cache}] \leftarrow \{\mathbf{K}_p, \mathbf{V}_p\}$ ▷ Update prompt KV cache
`/* Retrieve response features from cache */`
$\{\mathbf{K}_r, \mathbf{V}_r\} \leftarrow \mathcal{C}_r[l][\texttt{kv\_cache}]$ ▷ Reuse cached response KV
`/* Compute attention with mixed features */`
$\mathbf{K} \leftarrow [\mathbf{K}_p; \mathbf{K}_r]$; $\mathbf{V} \leftarrow [\mathbf{V}_p; \mathbf{V}_r]$
$\mathbf{AttnOut}_p \leftarrow \text{Attention}(\mathbf{Q}_p, \mathbf{K}, \mathbf{V})$ ▷ Only compute prompt attention
$\mathcal{C}_p[l][\texttt{attn}] \leftarrow \mathbf{AttnOut}_p$ ▷ Update prompt attention cache
$\mathbf{AttnOut}_r \leftarrow \mathcal{C}_r[l][\texttt{attn}]$ ▷ Reuse cached response attention
$\mathbf{AttnOut} \leftarrow [\mathbf{AttnOut}_p; \mathbf{AttnOut}_r]$ ▷ Combine prompt and response attention
$\mathbf{h} \leftarrow \mathbf{x}_{in} + \mathbf{AttnOut}$ ▷ Post-attention residual (using layer input $\mathbf{x}_{in}$)
`/* -- FFN Block -- */`
$\mathbf{h}_p, \mathbf{h}_r \leftarrow \mathbf{h}_{1:|\mathbf{c}|}, \mathbf{h}_{|\mathbf{c}|+1:}$ ▷ Split post-attention state
$\mathbf{h}_{p\_norm} \leftarrow \text{LayerNorm}(\mathbf{h}_p)$
$\mathbf{FFNOut}_p \leftarrow \text{FFN}(\mathbf{h}_{p\_norm})$ ▷ Compute FFN for prompt
$\mathcal{C}_p[l][\texttt{mlp}] \leftarrow \mathbf{FFNOut}_p$ ▷ Update prompt FFN cache
$\mathbf{FFNOut}_r \leftarrow \mathcal{C}_r[l][\texttt{mlp}]$ ▷ Reuse cached response FFN
$\mathbf{FFNOut} \leftarrow [\mathbf{FFNOut}_p; \mathbf{FFNOut}_r]$ ▷ Combine FFN outputs
$\mathbf{x}_{out} \leftarrow \mathbf{h} + \mathbf{FFNOut}$ ▷ Final output for this layer
return $\mathbf{x}_{out}, \mathcal{C}_p, \mathcal{C}_r$ ▷ Return layer output and updated caches

Algorithm 5 dLLM-Cache: Case 3 - Refresh Response Only

Require: Layer input $\mathbf{x}_{in}$, layer index l, caches $\mathcal{C}_p$ and $\mathcal{C}_r$

▷ $\mathbf{x}_{in}$ is the output of layer $l-1$

/* Case 3: Refresh response only, reuse prompt features */

$\mathbf{x}_{r\_in} \leftarrow \mathbf{x}_{in,|\mathbf{c}|+1:}$ ▷ Layer's response input part

/* Retrieve prompt features from cache */

$\{\mathbf{K}_p, \mathbf{V}_p\} \leftarrow \mathcal{C}_p[l][\texttt{kv\_cache}]$ ▷ Reuse cached prompt KV

$\mathbf{AttnOut}_p \leftarrow \mathcal{C}_p[l][\texttt{attn}]$ ▷ Reuse cached prompt attention

$\mathbf{FFNOut}_p \leftarrow \mathcal{C}_p[l][\texttt{mlp}]$ ▷ Reuse cached prompt FFN

/* Compute fresh response features */

$\mathbf{x}_{r\_norm} \leftarrow \text{LayerNorm}(\mathbf{x}_{r\_in})$

$\mathbf{Q}_r \leftarrow \text{Q\_proj}(\mathbf{x}_{r\_norm})$; $\mathbf{K}_r \leftarrow \text{K\_proj}(\mathbf{x}_{r\_norm})$; $\mathbf{V}_r \leftarrow \text{V\_proj}(\mathbf{x}_{r\_norm})$

$\mathcal{C}_r[l][\texttt{kv\_cache}] \leftarrow \{\mathbf{K}_r, \mathbf{V}_r\}$ ▷ Update response KV cache

/* Compute attention with mixed features */

$\mathbf{K} \leftarrow [\mathbf{K}_p; \mathbf{K}_r]$; $\mathbf{V} \leftarrow [\mathbf{V}_p; \mathbf{V}_r]$

$\mathbf{AttnOut}_r \leftarrow \text{Attention}(\mathbf{Q}_r, \mathbf{K}, \mathbf{V})$ ▷ Only compute response attention

$\mathcal{C}_r[l][\texttt{attn}] \leftarrow \mathbf{AttnOut}_r$ ▷ Update response attention cache

$\mathbf{AttnOut} \leftarrow [\mathbf{AttnOut}_p; \mathbf{AttnOut}_r]$ ▷ Combine prompt and response attention

$\mathbf{h} \leftarrow \mathbf{x}_{in} + \mathbf{AttnOut}$ ▷ Post-attention residual (using layer input $\mathbf{x}_{in}$)

/* -- FFN Block -- */

$\mathbf{h}_p, \mathbf{h}_r \leftarrow \mathbf{h}_{1:|\mathbf{c}|}, \mathbf{h}_{|\mathbf{c}|+1:}$ ▷ Split post-attention state

/* Retrieve prompt FFN, Compute response FFN */

$\mathbf{h}_{r\_norm} \leftarrow \text{LayerNorm}(\mathbf{h}_r)$

$\mathbf{FFNOut}_r \leftarrow \text{FFN}(\mathbf{h}_{r\_norm})$ ▷ Compute FFN for response

$\mathcal{C}_r[l][\texttt{mlp}] \leftarrow \mathbf{FFNOut}_r$ ▷ Update response FFN cache

$\mathbf{FFNOut} \leftarrow [\mathbf{FFNOut}_p; \mathbf{FFNOut}_r]$ ▷ Combine FFN outputs

$\mathbf{x}_{out} \leftarrow \mathbf{h} + \mathbf{FFNOut}$ ▷ Final output for this layer

return $\mathbf{x}_{out}, \mathcal{C}_p, \mathcal{C}_r$ ▷ Return layer output and updated caches

Algorithm 6 dLLM-Cache: Case 4 - Adaptive Update

Require: Layer input $\mathbf{x}_{in}$, layer index l, caches $\mathcal{C}_p$ and $\mathcal{C}_r$, adaptive update ratio ρ

/* Case 4: Adaptive partial update when no refresh required */
/* Retrieve cached prompt features */
$\{\mathbf{K}_p, \mathbf{V}_p\} \leftarrow \mathcal{C}_p[l][\texttt{kv\_cache}]$
$\mathbf{AttnOut}_p \leftarrow \mathcal{C}_p[l][\texttt{attn}]$
$\mathbf{FFNOut}_p \leftarrow \mathcal{C}_p[l][\texttt{mlp}]$
if $\rho > 0$ **then** ▷ Only proceed if adaptive update is enabled
  /* Compute current response Value projections */
  $\mathbf{x}_{r\_in} \leftarrow \mathbf{x}_{in,|\mathbf{c}|+1:}$ ▷ Layer's response input part
  $\mathbf{x}_{r\_norm} \leftarrow \text{LayerNorm}(\mathbf{x}_{r\_in})$
  $\mathbf{V}_r^{\text{new}} \leftarrow \text{V\_proj}(\mathbf{x}_{r\_norm})$
  /* Retrieve cached response features */
  $\{\mathbf{K}_r, \mathbf{V}_r\} \leftarrow \mathcal{C}_r[l][\texttt{kv\_cache}]$
  /* Compute similarity to identify tokens needing update */
  for each token index j in response sequence **do**
    $s_j \leftarrow \frac{(\mathbf{V}_r^{\text{new}}[j])^\top \mathbf{V}_r[j]}{\|\mathbf{V}_r^{\text{new}}[j]\| \|\mathbf{V}_r[j]\|}$ ▷ Cosine similarity
  end for
  $\mathcal{I}_{\text{update}} \leftarrow$ indices of $\lfloor \rho |\mathbf{y}^{(k)}| \rfloor$ tokens with lowest s_j
  /* Selective computation for selected tokens */
  $\mathbf{x}_{r\_norm\_selected} \leftarrow$ gather tokens from $\mathbf{x}_{r\_norm}$ at indices $\mathcal{I}_{\text{update}}$
  $\mathbf{Q}_r^{\text{selected}} \leftarrow \text{Q\_proj}(\mathbf{x}_{r\_norm\_selected})$
  $\mathbf{K}_r^{\text{selected}} \leftarrow \text{K\_proj}(\mathbf{x}_{r\_norm\_selected})$
  /* Update KV cache with new values */
  $\mathbf{K}_r^{\text{updated}} \leftarrow \text{ScatterUpdate}(\mathbf{K}_r, \mathcal{I}_{\text{update}}, \mathbf{K}_r^{\text{selected}})$ ▷ Uses scatter
  $\mathcal{C}_r[l][\texttt{kv\_cache}] \leftarrow \{\mathbf{K}_r^{\text{updated}}, \mathbf{V}_r^{\text{new}}\}$ ▷ Always use new V
  /* Compute attention only for selected tokens */
  $\mathbf{K} \leftarrow [\mathbf{K}_p; \mathbf{K}_r^{\text{updated}}]$; $\mathbf{V} \leftarrow [\mathbf{V}_p; \mathbf{V}_r^{\text{new}}]$
  $\mathbf{AttnOut}_r^{\text{selected}} \leftarrow \text{Attention}(\mathbf{Q}_r^{\text{selected}}, \mathbf{K}, \mathbf{V})$
  /* Update response attention cache at selected positions */
  $\mathbf{AttnOut}_r \leftarrow \mathcal{C}_r[l][\texttt{attn}]$
  $\mathbf{AttnOut}_r^{\text{updated}} \leftarrow \text{ScatterUpdate}(\mathbf{AttnOut}_r, \mathcal{I}_{\text{update}}, \mathbf{AttnOut}_r^{\text{selected}})$
  $\mathcal{C}_r[l][\texttt{attn}] \leftarrow \mathbf{AttnOut}_r^{\text{updated}}$
  $\mathbf{AttnOut} \leftarrow [\mathbf{AttnOut}_p; \mathbf{AttnOut}_r^{\text{updated}}]$ ▷ Combine attn outputs
  $\mathbf{h} \leftarrow \mathbf{x}_{in} + \mathbf{AttnOut}$ ▷ Post-attention residual (using layer input $\mathbf{x}_{in}$)
  /* -- FFN Block (Adaptive) -- */
  $\mathbf{h}_p, \mathbf{h}_r \leftarrow \mathbf{h}_{1:|\mathbf{c}|}, \mathbf{h}_{|\mathbf{c}|+1:}$ ▷ Split post-attention state
  /* Gather selected tokens from response post-attention state */
  $\mathbf{h}_r^{\text{selected}} \leftarrow$ gather tokens from $\mathbf{h}_r$ at indices $\mathcal{I}_{\text{update}}$
  /* Compute FFN only for selected tokens */
  $\mathbf{h}_{r\_selected\_norm} \leftarrow \text{LayerNorm}(\mathbf{h}_r^{\text{selected}})$
  $\mathbf{FFNOut}_r^{\text{selected}} \leftarrow \text{FFN}(\mathbf{h}_{r\_selected\_norm})$
  /* Update response FFN cache at selected positions */
  $\mathbf{FFNOut}_r \leftarrow \mathcal{C}_r[l][\texttt{mlp}]$
  $\mathbf{FFNOut}_r^{\text{updated}} \leftarrow \text{ScatterUpdate}(\mathbf{FFNOut}_r, \mathcal{I}_{\text{update}}, \mathbf{FFNOut}_r^{\text{selected}})$
  $\mathcal{C}_r[l][\texttt{mlp}] \leftarrow \mathbf{FFNOut}_r^{\text{updated}}$
  $\mathbf{FFNOut} \leftarrow [\mathbf{FFNOut}_p; \mathbf{FFNOut}_r^{\text{updated}}]$ ▷ Combine FFN outputs
else ▷ Case: $\rho = 0$
  /* Pure cache retrieval - no updates */
  $\mathbf{AttnOut}_r \leftarrow \mathcal{C}_r[l][\texttt{attn}]$
  $\mathbf{AttnOut} \leftarrow [\mathbf{AttnOut}_p; \mathbf{AttnOut}_r]$
  $\mathbf{h} \leftarrow \mathbf{x}_{in} + \mathbf{AttnOut}$ ▷ Post-attention residual
  $\mathbf{FFNOut}_r \leftarrow \mathcal{C}_r[l][\texttt{mlp}]$
  $\mathbf{FFNOut} \leftarrow [\mathbf{FFNOut}_p; \mathbf{FFNOut}_r]$ ▷ Combine FFN outputs
end if
$\mathbf{x}_{out} \leftarrow \mathbf{h} + \mathbf{FFNOut}$ ▷ Final output for this layer
return $\mathbf{x}_{out}, \mathcal{C}_p, \mathcal{C}_r$ ▷ Return layer output and updated caches