Implementation of Two Compute-Skipping Policies in dLLM-v2

Oytun Kuday Duran

June 13, 2026

1 Introduction

Large language models with iterative denoising/generation pipelines (such as Fast-dLLM v2) can perform repeated computations across adjacent denoising steps. In practice, many hidden representations change only slightly from one step to the next, which creates an opportunity to reduce computation by reusing previously computed outputs instead of recomputing them. This programming assignment explores that opportunity through *compute-skipping* policies based on cosine similarity of hidden states.

The main idea is simple: if the hidden-state input to a computation unit at the current denoising step is highly similar to the corresponding hidden-state input from the previous denoising step, then the new output is likely to be close to the previous output. In this case, we can skip the computation and reuse the cached output, potentially reducing FLOPs while preserving model accuracy. We study Token-level skipping and Layer-level skipping policies on top of the Fast-dLLM v2 codebase.

The goal of this homework is to implement these two policies and analyze the trade-off between efficiency and quality. Specifically, in terms of **accuracy vs. FLOPs**, showing how skipping affects model performance. This evaluation helps answer an important systems question: *How much computation can be safely skipped before accuracy degrades significantly?*

2 Layer-Level Skipping

Layer-level skipping is implemented with a cache-reuse optimization for decoder layers during denoising steps. For each layer, we cache the previous input hidden states and the corresponding layer output. On the next upcoming denoising step, we check the cosine similarity of the current input to the cached input, and if they are sufficiently similar, we skip the layer computation and reuse the cached output. This reduces compute when hidden states entering the layer have change only slightly across adjacent denoising steps.

3 Token-Level Skipping

Token-level skipping extends layer-level reuse by making the decision at token granularity of hidden states instead of skipping an entire decoder layer. For each layer and denoising step, we compare the current token hidden states with cached token hidden states from the previous denoising step, and recompute only unstable tokens that exhibit low cosine similarity. For stable tokens (high cosine similarity), we reuse cached outputs, while recomputed outputs for low-similarity tokens are scattered back into a full tensor.

At a high level, the implementation maintains per-layer caches for both the attention and MLP subpaths. For attention, it stores the normalized input to attention and the full attention output. For the MLP, it stores the normalized MLP input (after post-attention layer normalization) and the full MLP output. These caches are keyed by layer index and are reused only when the execution context is compatible. To guard correctness, the manager resets caches on commit passes and the execution context changes, using a context signature that includes hidden-state shape, past sequence length, and an attention-mask signature. This is more robust than shape-only checks because token outputs depend on the prompt/past KV length and masking structure, not just tensor dimensions. I do not reuse or merge tokens across different batch elements even if their hidden states appear very similar.

Adaptive Reuse Policy. Token selection is driven by cosine similarity between the current normalized hidden states and the cached normalized hidden states from the previous denoising step. The function computes similarity per token along the hidden dimension and then uses an adaptive reuse policy. It first computes a batch-level average cosine similarity and maps that average to a reuse ratio. It then applies that ratio per sample row by selecting the lowest-similarity tokens for recomputation. This design preserves batch structure while still adapting the amount of recomputation to the observed stability of the denoising step. Forced(masked) tokens (from generator.py) are merged into the recompute mask before filling the remaining budget with lowest-similarity tokens to ensure that they're calculated.

$$r(\texttt{avg\_cos}) = \begin{cases} \alpha \cdot 0.80, & \text{if } \texttt{avg\_cos} \geq 0.995, \\ \alpha \cdot 0.60, & \text{if } 0.99 \leq \texttt{avg\_cos} < 0.995, \\ \alpha \cdot 0.40, & \text{if } 0.98 \leq \texttt{avg\_cos} < 0.99, \\ 0.00, & \text{otherwise.} \end{cases}$$

Here, $r(\texttt{avg\_cos})$ denotes the adaptive *reuse ratio* (i.e., the fraction of tokens reused from cache. The tokens with highest cosine similarities are selected.), and $\alpha \in (0, 1]$ is a scaling factor that controls how aggressively reuse is applied across all similarity regimes. I empirically find that if I try to subsample when avg. cos. is lower than 98%, the model starts performing significantly worse. I believe the reason is the importance of cross-token interactions in attention before the hidden states start to converge.

Why not use a simple per-token threshold? The first approach comes to mind is to reuse every token whose cosine similarity exceeds a fixed threshold and recompute only the rest. In practice, this was not robust or efficient in our batched dLLM setting. First, thresholding produces a highly variable number of recomputed tokens across batches, samples and layers, which creates ragged computation patterns. For example, one sample in a batch may require recomputing only a few tokens while another may require many, making it difficult to maintain execution. This is why the implementation uses adaptive reuse policy and a validity mask. This way, we are preserving per-sample structure, and it simplifies attention-mask gathering, prompt-KV alignment, and scattering-back to the correct positions. Moreover, a threshold-only policy can select too few tokens, in which case the overhead of gather/scatter, mask slicing, and partial-kernel setup may dominate the compute saved.

Stability of token reusing. Empirically, I find that when the average cosine similarity of a layer is below roughly 0.98, continuing to subsample aggressively for reuse often caused the hidden-state trajectory to drift and generation quality to collapse. This behavior is consistent with the iterative nature of denoising: when representations are not yet stable, reuse errors accumulate instead of dissipating. Therefore, the token-selection policy is made *stability-aware*: I use the batch-average cosine similarity as a coarse indicator of whether the current denoising step is in a high-stability regime. When average similarity is high, we increase reuse; when average similarity is lower, we reduce reuse or recompute all tokens.

The attention path uses query-row selective recomputation. After applying the input layer norm, the wrapper checks whether a valid attention cache exists. If not, it enters a warmup path and recomputes full attention. Otherwise, it builds a query recompute mask indicating which token positions should recompute their attention outputs. It computes current K/V for all tokens in the block, but computes Q only for the selected query positions. This asymmetry is intentional: the output for a selected query depends on all keys and values, so K/V must remain current for correctness, while Q and the expensive attention rows can be decreased.

The MLP path is simpler because it is token-wise and does not involve cross-token interactions. After the attention residual is applied, the wrapper applies post-attention normalization, compares it against the cached MLP input, and constructs a token recompute mask using the same cosine-similarity-based selection logic (Here, it is easier to use a directly-threshold based approach, but for convenience, I am applying the exact same policy). It then gathers only the selected tokens, runs the MLP on those tokens, and scatters the results back into a full tensor using the cached MLP output as the base. Unselected tokens directly reuse their cached MLP outputs. Because the MLP output for a token depends only on that token's hidden state, this selective recomputation is much more straightforward than in attention and offers a clean source of compute savings.

4 Results

Experimental Setup. For the experiments, I use NVIDIA H200 GPUs (144GB HBM). Machines are equipped with AMD EPYC processors and connected via NVLink where applicable, running Ubuntu 22.04, Python 3.13. with CUDA 12.6. Everything about model and evaluation configurations are the default settings as provided in the repository.

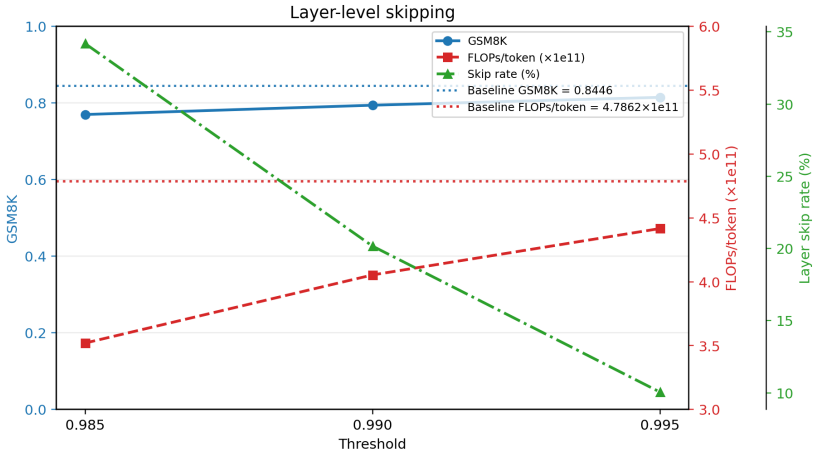

Figure 1: Layer-level skipping on GSM8K. The baseline (no skipping) is included at skip rate 0%. The left y-axis shows GSM8K accuracy, and the right y-axis shows FLOPs/token scaled by 10^{11}. Increasing the layer-level skip rate reduces FLOPs/token, but also decreases accuracy.

Experiments. For the layer-level skipping experiments, I sweep the cosine-similarity threshold in the range [0.985, 0.995]. For token-level skipping, I evaluate multiple adaptive reuse-ratio per similarity threshold: (a) 80%–60%–40%, (b) 90%–80%–10%, (c) 90%–80%–60%, and (d) 90%–80%–0%. I use the default configurations of model and evaluator in the repository, and GSM8K is chosen for the benchmark. FLOPs are counted using *PyTorch*'s built-in utils in the original repository's evaluation loop, accuracy is collected from evaluator's output, and skipped vs computed tokens/layers are counted internally.

As shown in Figures 1 and 2, layer-level skipping yields a clearer accuracy–FLOPs trade-off while token-level skipping is more sensitive. In particular, aggressive token reuse can reduce the amount of recomputation but also introduces gather/scatter, mask slicing, and partial-attention setup overheads, which may reduce or even offset the expected compute savings when only a small number of tokens are recomputed. Moreover, in dLLM denoising, reused tokens that do not re-enter attention still influence other tokens through the attention context, which can delay convergence of the generated output, require more calls to finish, and in some cases produce unstable or nonsensical behavior. This is con-

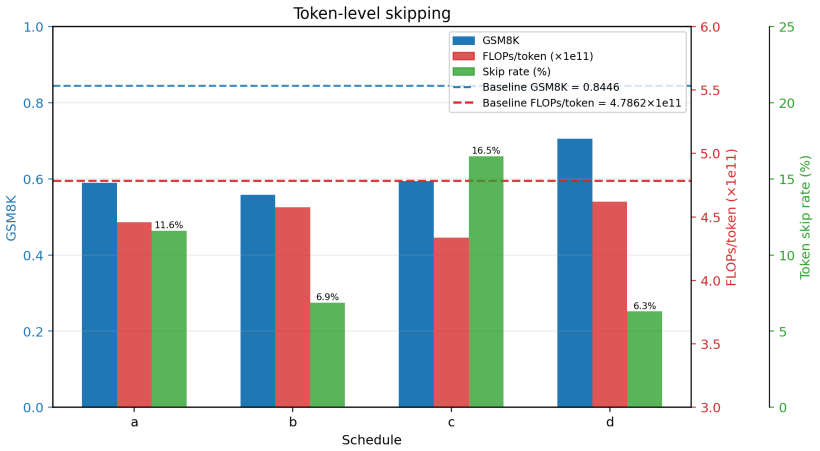

Figure 2: Token-level skipping on GSM8K using adaptive reusing policy. The blue axis shows GSM8K accuracy, the red axis shows FLOPs/token (scaled by 10^{11}), and the green axis shows layer skip rate (%). Dashed reference lines indicate the no-skipping baseline for accuracy and FLOPs/token.

sistent with our accuracy graphs vs the 3rd reuse ratio (in my policy) between cases **a to c** and **d**. Before all hidden states start to converge (average cosine similarity is low), even if we try to reuse only a few tokens from the previous states, the convergence slows down and the accuracy drop is significant. Even if we try to reuse the most similar 10% tokens when the average cosine similarity of the layer input is between 0.98% and 0.99%, we lose nearly 10% accuracy.