SPARSEDECODING: DECODING-AWARE PRUNING FOR ACCURATE AND EFFICIENT LLM INFERENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

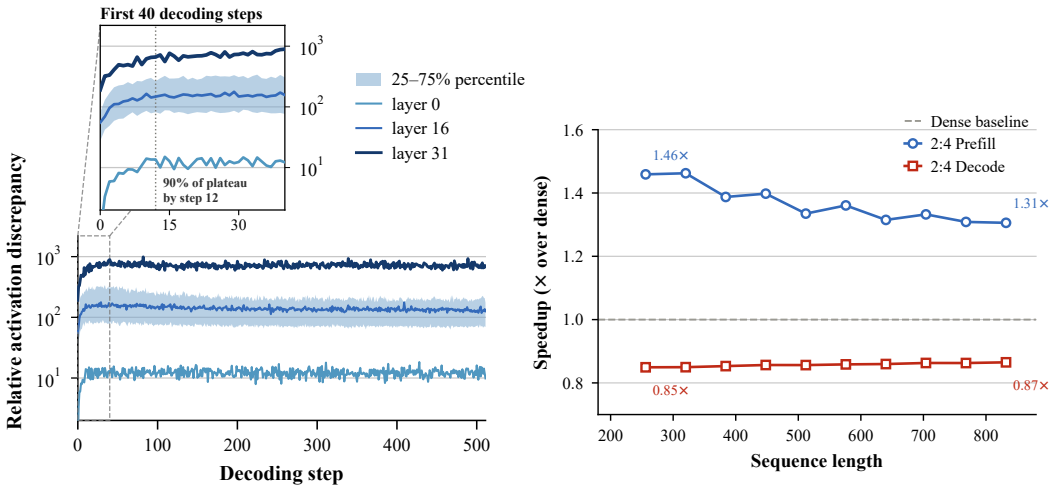
The memory-bound nature of the decoding stage of large language model (LLM) inference incurs significant latency. Layer-wise training-free network pruning approaches guided by the Hessian have been a prominent solution to this problem, as pruning reduces the number of nonzero parameters read from memory during decoding. Nevertheless, typical methods in this line compute the Hessian using pre-collected natural sequences, whereas the model is fed self-generated tokens during decoding, creating a distribution shift between the two sequences. The Hessian calculated on the natural sequence is different from that calculated on the generated sequence. We observe that this discrepancy causes the activation distribution during generation to deviate from that used for pruning, further hurting the pruned model performance. Moreover, most existing LLM pruning methods that bring actual speedup primarily target the sparse matrix-matrix (SpMM) multiplication, providing limited support for the sparse matrix-vector (SpMV) operations, which dominate decoding. To solve these problems, we introduce **SparseDecoding**, a principled *decoding-aware* pruning framework tailored for accurate and efficient LLM decoding. Specifically, at the algorithmic axis, SparseDecoding constructs calibration matrices from layer-wise activations collected during the dense-model autoregressive generation, excluding prefill, thereby aligning the pruning objective with the decoding activations. At the system axis, we develop an optimized $N{:}M$ sparse matrix-vector kernel with bitmask indexing and fixed-step traversal. Substantial empirical results on representative LLMs (Llama-3.1-8B, Llama-3.3-70B, Qwen3-14B / 32B) demonstrate that our method consistently outperforms standard fixed-text calibration on the long-form generation benchmarks while achieving up to $1.48\times$ end-to-end wall-clock decoding speedup on A100 GPUs.

1 INTRODUCTION

Large language models (LLMs) are increasingly used for long-output applications such as document drafting, long-form content generation, and code generation (Bai et al., 2025; Wu et al., 2025; Chen et al., 2021; Jain et al., 2025). The autoregressive decoding process of LLMs is memory-bound and typically dominates end-to-end inference latency (Kim et al., 2025; Fu et al., 2024). Generating such long outputs incurs substantial latency, making inference efficiency a practical concern for deployment. Weight pruning addresses the inference cost problem of neural networks by zeroing out insignificant weights (Han et al., 2015). In terms of LLMs, due to the compute cost and data restrictions, training-free methods are the dominant ones (Frantar & Alistarh, 2023; Su & Wang, 2026) for sparsifying LLMs, especially those based on the OBS framework (Hassibi & Stork, 1992).

Existing training-free pruning methods (Frantar & Alistarh, 2023; Su & Wang, 2026) typically rely on a small calibration set to estimate or compensate for the output reconstruction error. Consequently, calibration data play a critical role in determining the quality of the pruned model. Existing methods usually collect calibration activations by running fixed sequences from the calibration set under teacher forcing. This protocol is appropriate for static-input inference settings, where the model inputs remain unchanged by its own predictions. However, autoregressive decoding continuously extends the input context with previously generated tokens. Pruning errors at early decoding steps may therefore alter subsequent predictions and contexts, causing the resulting activation distribution to progressively deviate from that observed during teacher-forced calibration. As shown

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



(a) Activation discrepancy during decoding.

(b) Prefill and decoding speedup.

Figure 1: Motivating observations for decoding-aware sparse inference. (a) The relative activation discrepancy between the dense and pruned models grows rapidly during the first few decoding steps before settling into a plateau. The shaded region shows the interquartile range across transformer layers, while the curves correspond to the **first** (layer 0), **middle** (layer 16), and **last** (layer 31) layers. All layers reach 90% of their plateau by decoding step 12, after which the activation discrepancy remains largely unchanged. (b) Speedup of the 2:4 sparse Tensor Core path over dense execution. Existing sparse kernels provide 1.31 to 1.46× speedups during the prefill stage, but achieve only 0.85 to 0.87× of dense performance during autoregressive decoding. Results are obtained on Llama-3.1-8B with 2:4 sparsity on an NVIDIA A100 GPU.

in Figure 1a, the relative activation discrepancy increases rapidly during the early stage of autoregressive decoding, with all layers reaching at least 90% of their eventual plateau within the first 12 decoding steps. After this initial increase, the discrepancy remains largely stable throughout the subsequent decoding process, even as generation continues for hundreds of additional steps. This behavior suggests that the activation distribution during decoding differs from that observed under fixed teacher-forced calibration, motivating us to construct the calibration objective directly from decode-time activations.

Meanwhile, reducing the theoretical computational cost of pruning alone does not necessarily translate into lower latency, and achieving practical speedups requires efficient sparse-kernel implementations. We consider N:M semi-structured sparsity as a case study, since it offers hardware-friendly regularity while retaining flexibility in selecting weights (Fang et al., 2024). Existing 2:4 Sparse Tensor Core libraries, such as cuSPARSELt (NVIDIA Corporation, 2021) are optimized primarily for Sparse Matrix-Matrix Multiplication (SpMM) (Lin et al., 2023). By contrast, operations during decoding are dominated by Sparse Matrix-Vector Multiplication (SpMV) (Hong et al., 2024), rather than SpMM, and thus cannot be accelerated by cuSPARSELt implementations.

In this paper, we introduce **SparseDecoding** (Figure 2) to address these two issues. On the algorithmic side, we build the calibration matrices from layer-wise input activations collected as the dense model generates tokens autoregressively. We keep only the activations from decoding and discard those from prefill. This decoding-aware calibration can be used directly with existing pruning methods such as SparseGPT (Frantar & Alistarh, 2023). On the system side, inspired by nmSPARSE (Lin et al., 2023), a GPU library for general N:M sparse computation, we design a customized N:M SpMV kernel for efficient autoregressive decoding. Instead of storing a separate 32-bit column index for each nonzero, we represent the sparsity pattern with bitmasks, using one bit per input column and packing every 32 consecutive bits into a 32-bit mask. For the 50% N:M patterns with $M \leq 32$ considered in our kernel design, each aligned 32-bit mask word contains exactly 16 set bits. This allows every output row to traverse its bitmask in the same number of steps, keeping the workload uniform across GPU threads. Together, decoding-aware calibration and the optimized N:M SpMV kernel preserve generation quality while providing practical end-to-end decoding speedups.

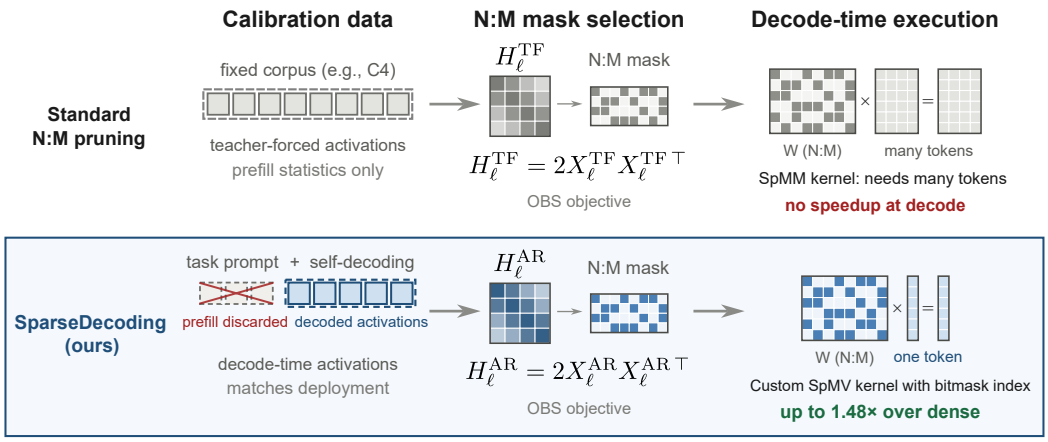


Figure 2: **Overview of SparseDecoding.** Unlike standard pruning methods collecting calibration activations from fixed corpora, SparseDecoding calibrates on the model-generated tokens from task prompts while discarding the prefill activations. Meanwhile, existing $N{:}M$ sparse Tensor Core kernels accelerate prefill but often underperform dense execution during autoregressive decoding. SparseDecoding instead introduces a decoding-oriented sparse execution pipeline that converts pruned weights into an efficient $N{:}M$ representation and employs an optimized sparse decoding kernel, achieving up to $1.48\times$ decoding speedup under the experimental setup described in Sec. 4.

Our contributions are as follows:

1. We propose a decoding-aware pruning method that aligns the pruning objective with the activations encountered during autoregressive decoding.
2. We further develop an optimized $N{:}M$ SpMV kernel for efficient sparse decoding, using bitmask indexing and fixed-step traversal to reduce metadata and indexing overhead.
3. Extensive experiments across multiple LLMs show that SparseDecoding improves long-form generation quality over existing pruning baselines while achieving up to $1.48\times$ end-to-end decoding speedup on A100 GPUs.

2 Related Work

2.1 One-Shot Pruning

Network pruning is widely used to reduce the size and inference cost of pretrained models by eliminating redundant weights (LeCun et al., 1989; Han et al., 2015; 2016; Hoefler et al., 2021). In terms of pruning granularity, existing methods can be broadly categorized into structured (Ma et al., 2023; Wang et al., 2023), unstructured (Han et al., 2015; Singh & Alistarh, 2020), and semi-structured pruning (Fang et al., 2024; Mishra et al., 2021; Zhang et al., 2022). Structured pruning eliminates entire parameter groups, such as filters or channels (He et al., 2017; Ma et al., 2023), while preserving regular computation that can be efficiently handled by dense hardware and software kernels (Mishra et al., 2021), but offering less flexibility in selecting individual weights (Hoefler et al., 2021). Unstructured pruning provides greater flexibility, but its irregular nonzero locations introduce indirect memory access, limited parallelism, and scheduling overhead (Han et al., 2015; Wen et al., 2016). Semi-structured pruning lies between these two extremes. A representative example is $N{:}M$ sparsity (Pool & Yu, 2021), which retains N nonzero weights in every group of M, balancing hardware efficiency with flexible weight selection.

2.2 Calibration Data for Pruning

One-shot pruning methods (e.g., SparseGPT (Frantar & Alistarh, 2023), Wanda (Sun et al., 2024), ROSE (Su & Wang, 2026)) typically use a small calibration set, such as C4 (Raffel et al., 2020) or WikiText (Merity et al., 2017), to estimate weight importance. Recent studies have shown that the

choice of calibration data can substantially affect pruning quality. By comparing multiple pretraining and downstream datasets, Bandari et al. (2024) show that C4 is often not the best calibration source across tasks. Ji et al. (2025) further show that self-generated calibration data better aligned with the model's distribution can improve pruning performance. These studies focus on which calibration corpus to use, whereas SparseDecoding changes how calibration activations are collected specifically from the autoregressive decoding generations. More closely related to our setting, RAC (Lucas et al., 2026) uses both input activations and activations from the model's on-policy chain-of-thought trajectory for layer-wise reconstruction, while RESP (Wang et al., 2025) uses self-generated reasoning traces with a decode-only gradient objective to estimate importance for structured pruning. Both methods incorporate decode-time information into pruning, while we focus on how the calibration objective differs from the actual decode-time reconstruction objective. SparseDecoding builds the calibration matrix from task-conditioned decode-step activations, excluding prefill, and uses it to define the layer-wise reconstruction objective. Theorem A.1 measures the exact worst-case relative discrepancy between the calibration and decode-time reconstruction objectives.

2.3 Sparse GPU Kernels and Efficient Decoding

Whether a sparse model is faster in practice depends on how the sparse pattern is executed on GPUs. General sparse kernels such as Sputnik (Gale et al., 2020) show that locality and load balancing can improve sparse matrix computation in deep learning, but irregular sparsity also introduces metadata, indirect memory access, and scheduling overhead. NVIDIA cuSPARSELt (NVIDIA Corporation, 2021) targets structured sparse matrix-matrix computation and is more suitable for prefill or larger-batch GEMM/SpMM. In autoregressive decoding, however, the core linear layers at batch one or small batches are closer to GEMV/SpMV, so Tensor-Core-oriented $2{:}4$ paths do not directly cover our target setting. Lin et al. (2023) provide a general execution path for $N{:}M$ sparse weights by supporting both SpMV and SpMM. We therefore develop an $N{:}M$ SpMV execution path specialized for sparse decoding, using bitmask indexing and fixed-step traversal. Our implementation achieves up to a $1.48\times$ end-to-end decoding speedup on A100 GPUs.

3 Proposed Method: SparseDecoding

SparseDecoding is a pruning framework for layer-wise post-training pruning and can be combined with OBS-based solvers such as SparseGPT (Frantar & Alistarh, 2023). We first introduce the preliminaries on OBS and the relevant works (Sec. 3.1), and then elaborate on our proposed method (Sec. 3.2). Of note, the method is not only for more accurate pruning. We also propose a kernel design (Sec. 3.3) to achieve actual wall-clock speedup on hardware.

3.1 Preliminary

Layer-wise post-training pruning. Directly optimizing the entire model for post-training pruning is computationally prohibitive due to the large number of parameters. Therefore, a widely adopted strategy is to decompose the global compression problem into a sequence of layer-wise reconstruction problems (Frantar & Alistarh, 2022; Dong et al., 2017). For layer ℓ, given the layer-wise calibration activations X_ℓ and a layer-wise target sparsity S_ℓ, the goal is to find a sparse weight matrix $\widehat{W}_\ell$ that changes the layer output as little as possible on X_ℓ:

$$\widehat{W}_\ell = \underset{\widetilde{W}_\ell}{\operatorname{argmin}} \left\| \left(W_\ell - \widetilde{W}_\ell\right) X_\ell \right\|_F^2 \quad \text{s.t.} \quad \operatorname{sparsity}\left(\widetilde{W}_\ell\right) = S_\ell, \tag{1}$$

where $\widetilde{W}_\ell$ ranges over all weight matrices that satisfy the layer-wise sparsity constraint S_ℓ, $\widehat{W}_\ell$ denotes the sparse weight matrix that minimizes the objective, and $\|\cdot\|_F^2$ denotes the squared Frobenius norm. Solving this problem for each layer sequentially and applying the layer-wise sparse weights yields the final pruned network.

Optimal Brain Surgeon for layer-wise pruning. The Optimal Brain Surgeon (OBS) framework (Hassibi & Stork, 1992) provides an efficient way to solve the layer-wise problem in Eq. (1). Its key observation is that, because the objective is an ℓ_2 norm, the problem decouples across the rows of W_ℓ into independent per-row subproblems. Within each row, OBS builds a second-order Taylor

approximation of the objective around the current weights, which requires the Hessian of the reconstruction error with respect to that row, $H = 2X_\ell X_\ell^\top$, a matrix shared by every row since it depends only on the calibration activations. This approximation admits a closed form for **(1)** identifying the least salient weight w_q in the row, whose removal induces the smallest increase in reconstruction error; **(2)** computing the optimal update δw to the row's surviving weights that compensates for removing it. The saliency $\mathcal{L}_q$ and the update δw are given by

$$\mathcal{L}_q = \frac{w_q^2}{2[H^{-1}]_{qq}}, \quad \delta w = -\frac{w_q}{[H^{-1}]_{qq}} H^{-1}_{:,q}. \tag{2}$$

Here, $[H^{-1}]_{qq}$ denotes the q-th diagonal entry of the inverse Hessian, and $H^{-1}_{:,q}$ denotes its q-th column. The procedure is applied iteratively, one weight at a time, until the target sparsity S_ℓ is reached. At each pruning step, removing a weight requires the corresponding inverse Hessian information to be updated. Performing a full matrix re-inversion after every removal is computationally prohibitive. SparseGPT (Frantar & Alistarh, 2023) addresses these challenges by fixing the pruning order in advance, enabling efficient, stable maintenance of the required inverse Hessian information.

3.2 Proposed Method: SparseDecoding

SparseDecoding is a decoding-aware pruning framework that aligns post-training pruning with the autoregressive states encountered during generation. In OBS (Hassibi & Stork, 1992), the calibration data affect which weights are removed and how the remaining weights are updated through the Hessian $H = 2X_\ell X_\ell^\top$. Standard calibration collects X_ℓ^{TF} from fixed text such as C4 (Raffel et al., 2020), where every token in the context is given in advance. Consequently, the Hessian summarizes activations under fixed contexts rather than the model-generated contexts.

However, autoregressive decoding differs from calibration on fixed text because later model states depend on the tokens generated at earlier steps. Errors introduced by pruning at early decoding steps can therefore affect subsequent token predictions and the contexts conditioned on by the model, with these effects accumulating over generation. As a result, the decode-time activation distribution can gradually diverge from that observed under fixed-text calibration. Thus, we first let $\mathcal{C} = \{q_m\}_{m=1}^K$ be a set of calibration prompts drawn from the target task. Rather than running fixed text through the model, we let the *dense* model produce its own continuation of each prompt,

$$y_t^{(m)} \sim P_\theta\big(\cdot \mid q_m,\, y_{<t}^{(m)}\big), \qquad t = 1, \ldots, T_m, \tag{3}$$

so that the context at every step is induced by the model's own previously generated outputs. For prompt q_m, we write $\mathcal{P}_m$ for the prompt positions consumed during prefill and $\mathcal{D}_m$ for the T_m positions emitted during decoding. For a linear layer ℓ with weights $W_\ell \in \mathbb{R}^{d_{\mathrm{out}} \times d_{\mathrm{in}}}$, let $x_{\ell,t}^{(m)} \in \mathbb{R}^{d_{\mathrm{in}}}$ denote its input activation at step t. We record $x_{\ell,t}^{(m)}$ for every prunable linear layer and keep only the steps $t \in \mathcal{D}_m$, discarding the activations produced while processing $\mathcal{P}_m$. We discard prefill positions because our calibration objective specifically targets the token-by-token, self-conditioned decoding regime. As shown in Figure 1a, the activation discrepancy grows rapidly during the early decoding steps, motivating our calibration on decode-time activations. Then we collect every retained activation to construct a matrix, with each activation being a column of the matrix,

$$X_\ell^{\mathrm{AR}} = \big[\, x_{\ell,t}^{(m)} \,\big]_{m=1,\ldots,K;\; t \in \mathcal{D}_m} \in \mathbb{R}^{d_{\mathrm{in}} \times N_{\mathrm{D}}}, \quad N_{\mathrm{D}} = \sum_{m=1}^{K} |\mathcal{D}_m|, \tag{4}$$

where N_{D} denotes the total number of decoding steps across the K prompts. During the model pruning phase, we therefore find the sparse weights $\widehat{W}_\ell$ that follow the minimization objective in Eq. (1) to minimize the influence on the layer output based on these activations. Written out, our specific autoregressive layer-wise pruning reconstruction objective is defined as:

$$\begin{aligned} \mathcal{L}_\ell^{\mathrm{AR}} &= \big\| (W_\ell - \widehat{W}_\ell)\, X_\ell^{\mathrm{AR}} \big\|_F^2 \\ &= \sum_{m=1}^{K} \sum_{t \in \mathcal{D}_m} \big\| (W_\ell - \widehat{W}_\ell)\, x_{\ell,t}^{(m)} \big\|_2^2, \end{aligned} \tag{5}$$

so the loss ranges over all N_{D} generated tokens. While fixed-text calibration on C4 (Raffel et al., 2020) or reference text constructs X_ℓ^{TF} from teacher-forced activations, our method instead constructs X_ℓ^{AR} from autoregressive decode-time activations. The pruning solver itself remains unchanged, but the resulting reconstruction objective is defined over activations that more closely reflect those encountered during generation, without requiring additional training or a new optimizer.

3.3 N:M SpMV Kernel Design

We first instantiate the framework under $N{:}M$ semi-structured sparsity for our method. The $N{:}M$ pattern alone does not determine decoding efficiency. However, as shown in Figure 1b, existing 2:4 Sparse Tensor Core kernels reach 1.31-$1.46\times$ speedup during prefilling but only achieve 0.85-$0.87\times$ of dense throughput during decoding. In this subsection, we explain why the same sparse weights accelerate the prefilling yet remain inefficient during decoding, and describe the decode-time specialized kernel we develop to address this.

Decoding is dominated by SpMV operations. In batch-one or small-batch autoregressive decoding, each linear layer processes only the hidden state of the current token, so its computation is closer to General Matrix-Vector Multiplication (GEMV) than large-batch General Matrix-Matrix Multiplication (GEMM). After pruning, the corresponding operation becomes $N{:}M$ SpMV. We therefore develop a decoding-specialized $N{:}M$ SpMV kernel inspired by nmSPARSE (Lin et al., 2023). Our kernel takes advantage of the regular $N{:}M$ sparsity pattern to reduce the metadata needed to locate nonzero weights and to make indices be extracted efficiently:

(1) Bitmask indexing. We first represent the sparsity pattern of each row with a bitmask, where each bit indicates whether the weight at the corresponding column is retained. Because the nonzero weights are stored in increasing column order, the set bits in the mask correspond to the stored weights in the same order, thus removing the need to store a separate column index or local offset for every nonzero. At 50% sparsity, this representation uses only 2 bits of metadata per nonzero, compared with 32 bits for the `int32` indices used by nmSPARSE and 8 bits for a `uint8` offset representation. The metadata size is therefore reduced by $16\times$ and $4\times$, respectively, which is especially useful during batch-one decoding, where SpMV is largely memory-bound.

(2) Fixed-step bitmask traversal. For a general sparse matrix, different rows may contain different numbers of nonzeros, so traversing their bitmasks requires different numbers of steps. With 50% $N{:}M$ sparsity and $M \leq 32$, each aligned 32-bit mask contains exactly 16 set bits. The number of traversal steps is therefore fixed for every row and known at compile time. We process each bitmask with a fixed-length loop that the compiler can unroll. At each step, a bit-scan operation finds the least significant set bit, which gives the column of the next nonzero, and that bit is then cleared before continuing. Because each row requires the same number of traversal steps, the workload remains uniform across GPU threads, which allows the $N{:}M$ bitmask representation to be traversed efficiently during SpMV. We provide further implementation details in Appendix C.

4 Experiments

We evaluate SparseDecoding by addressing three questions: **(1)** whether autoregressive calibration preserves quality on long writing tasks; **(2)** whether the same calibration procedure generalizes to code generation; and **(3)** whether the optimized SpMV kernel reduces decoding latency. Together, these experiments cover the quality and efficiency goals introduced in Sec. 3.

4.1 Experimental Setup

Models. We evaluate four open-weight instruction-tuned language models from two model families at two parameter scales: Llama-3.1-8B, Llama-3.3-70B (Grattafiori et al., 2024), and Qwen3-14B/32B (Yang et al., 2025). We prune only linear layers supported by the pruning backend, leaving all other components unchanged. The corresponding dense model serves as the quality reference throughout our experiments. We conduct all of our experiments on NVIDIA A100 80 GB GPUs.

Benchmarks. Our method is specifically designed for long decoding scenarios. Thus, here we employ two representative benchmarks, *WritingBench* (Wu et al., 2025) and *ClassEval* (Du et al., 2023), to evaluate our method against others: **WritingBench for writing.** *WritingBench* (Wu et al.,

Table 1: **Results on the WritingBench benchmark under 50% and $2{:}4$ sparsity.** SparseDecoding (ours) calibrates on autoregressive generations conditioned on LongWriter prompts; C4 uses fixed C4 sequences. Dense is the unpruned reference. Columns report the overall score, the six domain scores, and the three requirement scores for *style* (R1), *format* (R2), and *length* (R3); "C" indicates the corresponding category-specific score. The six domains are *Academic & Engineering* (D1), *Finance & Business* (D2), *Politics & Law* (D3), *Literature & Arts* (D4), *Education* (D5), and *Advertising & Marketing* (D6). All scores are on a scale of 1-10. Bold marks the better overall score between SparseDecoding and C4 within each sparsity setting, based on unrounded values.

Model	Sparsity	Method	Overall	Domains						Requirements					
				D1	D2	D3	D4	D5	D6	R1	C	R2	C	R3	C
Llama-3.1-8B	0%	Dense	3.70	3.7	3.6	3.5	3.3	4.1	4.3	3.8	3.9	3.7	4.3	3.8	3.6
	50%	C4	2.25	2.4	2.2	2.1	1.9	2.4	2.6	2.2	2.2	2.2	2.5	2.2	2.1
		SparseDecoding	**2.54**	2.7	2.5	2.5	2.0	2.7	3.0	2.5	2.5	2.5	2.9	2.5	2.4
	2:4	C4	1.34	1.5	1.3	1.3	1.1	1.4	1.6	1.3	1.3	1.3	1.4	1.3	1.2
		SparseDecoding	**1.52**	1.7	1.6	1.5	1.3	1.5	1.6	1.5	1.5	1.5	1.5	1.4	1.4
Llama-3.3-70B	0%	Dense	4.65	4.6	4.5	4.5	4.5	5.1	5.2	4.7	4.9	4.6	5.2	4.8	4.5
	50%	C4	3.82	3.7	3.8	3.7	3.5	4.1	4.3	3.8	4.0	3.8	4.3	3.8	3.7
		SparseDecoding	**4.01**	4.0	3.9	3.8	3.8	4.4	4.5	4.0	4.2	4.0	4.5	4.1	3.8
	2:4	C4	3.10	3.1	3.2	3.0	2.6	3.3	3.5	3.1	3.0	3.1	3.6	3.0	2.8
		SparseDecoding	**3.29**	3.2	3.3	3.0	3.1	3.7	3.8	3.3	3.4	3.2	3.6	3.3	2.9
Qwen3-14B	0%	Dense	5.97	6.0	5.8	6.1	5.7	6.3	6.1	6.1	6.4	6.0	6.8	6.0	6.3
	50%	C4	4.62	4.6	4.4	4.6	4.0	5.4	5.2	4.8	5.0	4.6	5.2	4.5	4.6
		SparseDecoding	**5.35**	5.3	5.3	5.4	4.9	5.9	5.6	5.5	5.8	5.4	6.2	5.4	5.6
	2:4	C4	1.85	2.0	1.8	1.7	1.6	2.0	2.3	1.9	1.9	1.8	1.7	1.9	1.7
		SparseDecoding	**4.18**	4.2	4.3	4.3	3.3	4.8	4.6	4.2	4.4	4.2	4.9	4.1	4.2
Qwen3-32B	0%	Dense	6.48	6.5	6.4	6.5	6.3	6.7	6.6	6.6	6.8	6.5	7.2	6.6	6.8
	50%	C4	5.34	5.2	5.2	5.4	4.9	5.9	5.8	5.5	5.7	5.3	6.0	5.3	5.4
		SparseDecoding	**6.09**	6.0	6.0	6.1	5.8	6.5	6.3	6.2	6.5	6.1	6.8	6.1	6.4
	2:4	C4	2.99	3.0	2.9	2.9	2.3	3.5	3.9	3.0	3.1	3.0	3.3	2.9	2.8
		SparseDecoding	**5.31**	5.2	5.3	5.4	4.7	5.8	5.7	5.4	5.6	5.3	6.0	5.3	5.5

2025) contains $1{,}000$ writing prompts belonging to 100 subdomains within 6 domain categories, which are *Academic & Engineering* **(D1)**, *Finance & Business* **(D2)**, *Politics & Law* **(D3)**, *Literature & Arts* **(D4)**, *Education* **(D5)**, and *Advertising & Marketing* **(D6)**. Each prompt also introduces three extra requirements as the evaluation criteria: *style, format*, and *length*. Following the original setup, we decode at the temperature of 0.7, top-k value of 20, and top-p value of 0.8, with maximum generation lengths of $16{,}000$ tokens. As the benchmark requires an additional LLM as the evaluation judge, we select DeepSeek-V4-Flash (DeepSeek-AI, 2026) as the judge. All reported scores are on a scale of 1-10, with *higher* scores indicating better performance.

ClassEval for coding. *ClassEval* (Du et al., 2023) contains 100 hand-written Python class generation tasks covering 410 methods, averaging 33.1 tests per class. Given a class skeleton and natural-language description, the model generates the entire class in a single pass. We report class-level Pass@1, the fraction of tasks for which the model's greedy generation passes its class-level tests.

Pruning configurations. We evaluate $2{:}4$ and $8{:}16$ $N{:}M$ sparsity, as well as 50% unstructured sparsity. All configurations retain 50% of the weights, which holds the nominal sparsity constant across the three patterns. The two $N{:}M$ settings differ only in the grouping constraint; for instance, a $2{:}4$ mask selects two weights from each group of four, while an $8{:}16$ mask selects eight weights from each group of sixteen. Unstructured pruning provides a reference without the local $N{:}M$ constraint. All models we use in our experiments are pruned only once after calibration.

Calibration configurations. The C4 configuration follows the common post-training pruning pipeline. It collects layer inputs from fixed C4 text (Raffel et al., 2020). SparseDecoding collects layer inputs from decode steps produced by the dense model. The prompts come from Long-

Writer (Bai et al., 2025) for the writing experiments and from LiveCodeBench problem statements (Jain et al., 2025) for the code experiments without solutions or reasoning traces. Each dense model generates its own continuation, so the activations we collect are conditioned on the model's own output rather than on reference text. Both C4 and SparseDecoding use 1M calibration tokens, with only decode-step tokens counted for SparseDecoding. All of the models operate in non-thinking mode during both calibration and evaluation. Both configurations pass their activations to the same SparseGPT backend. Comparisons within one specified sparsity pattern therefore fix the model, pruning backend, and retained weight fraction. We repeat this comparison under both unstructured 50% and semi-structured $2{:}4$ sparsity on Qwen3 models in Sec. 4.5.

4.2 Main Results: WritingBench

Table 1 reports WritingBench scores for the dense checkpoints alongside models pruned with C4 calibration and with SparseDecoding calibration on LongWriter (Bai et al., 2025) prompts.

Effect of decoding-aware calibration. SparseDecoding improves the overall WritingBench score across all models and sparsity settings. Under 50% unstructured sparsity, the gains range from 0.19 to 0.75 points across the four models. Under $2{:}4$ sparsity, the improvements remain modest for the two Llama models, increasing from 1.34 to 1.52 for Llama-3.1-8B and from 3.10 to 3.29 for Llama-3.3-70B. In contrast, the gains are substantially larger for the Qwen models, with the overall score increasing from 1.85 to 4.18 for Qwen3-14B and from 2.99 to 5.31 for Qwen3-32B. We also observe similar improvements across most writing domains and the requirement scores. Since each comparison fixes the model, pruning backend, retained weight fraction, and sparsity pattern, the results suggest that calibration activations play an important role in preserving generation quality after pruning.

4.3 Main Results: ClassEval

Table 2 reports class-level Pass@1 for the dense checkpoints alongside models pruned with C4 calibration and SparseDecoding calibration on LiveCodeBench (Jain et al., 2025) problem statements.

Comparisons over calibration methods. SparseDecoding is ahead across all eight model-sparsity pairs by 4 to 15 points. Comparing with C4 under $2{:}4$ sparsity, Pass@1 rises from 9.0 to 24.0% on Qwen3-32B, from 0.0 to 6.0% on Qwen3-14B, from 11.0 to 15.0% on Llama-3.3-70B, and from 0.0 to 4.0% on Llama-3.1-8B. Notably, under 2:4 sparsity, C4 calibration yields 0.0 on both Llama-3.1-8B and Qwen3-14B, whereas SparseDecoding recovers nonzero scores.

Table 2: **Results on the ClassEval benchmark under 50% and $2{:}4$ sparsity.**

Model	Sparsity	Method	Pass@1 (%) ↑
Llama-3.1-8B	0%	Dense	24.0
	50%	C4	2.0
		SparseDecoding	**10.0**
	2:4	C4	0.0
		SparseDecoding	**4.0**
Llama-3.3-70B	0%	Dense	33.0
	50%	C4	22.0
		SparseDecoding	**28.0**
	2:4	C4	11.0
		SparseDecoding	**15.0**
Qwen3-14B	0%	Dense	36.0
	50%	C4	22.0
		SparseDecoding	**27.0**
	2:4	C4	0.0
		SparseDecoding	**6.0**
Qwen3-32B	0%	Dense	34.0
	50%	C4	25.0
		SparseDecoding	**30.0**
	2:4	C4	9.0
		SparseDecoding	**24.0**

4.4 Empirical Decoding Speedup

Table 3 reports decoding throughput and speedup relative to the dense model across $N{:}M$ patterns from $2{:}4$ to $16{:}32$. All measurements use NVIDIA A100 GPUs and the GPT-Fast framework (PyTorch, 2023). On each model, all four patterns achieve nearly identical throughput, with speedups of $1.42\times$ on Llama-3.1-8B, $1.48\times$ on Llama-3.3-70B, $1.35\times$ on Qwen3-14B, and $1.45\times$ on Qwen3-32B. This is expected from our kernel design. Since all patterns retain 50% of the weights and use the same bitmask metadata, they incur identical memory traffic per row. In contrast to the $2{:}4$ Sparse Tensor Core path in Figure 1b, which runs slower than dense execution during decoding, our kernel turns every evaluated $N{:}M$ pattern into a practical decoding speedup.

Table 3: **End-to-end decoding throughput and speedup over the dense baseline**. All of the N:M patterns retain 50% of the weights, so differences reflect kernel efficiency rather than compression. Every evaluated pattern decodes faster than the dense baseline, with speedups ranging from 1.34 to $1.48\times$. Measurements use a context length of 512 and report the median after 50 warmup and 50 measured iterations on NVIDIA A100 GPUs with GPT-Fast.

N:M	Llama-3.1-8B		Llama-3.3-70B		Qwen3-14B		Qwen3-32B	
	Token/s	Speedup	Token/s	Speedup	Token/s	Speedup	Token/s	Speedup
Dense	96.26	1.00×	21.67	1.00×	39.76	1.00×	19.15	1.00×
2:4	136.30	1.42×	30.17	1.39×	53.18	1.34×	27.69	1.45×
4:8	136.43	1.42×	31.64	1.46×	53.86	1.35×	27.83	1.45×
8:16	136.89	1.42×	29.90	1.38×	53.42	1.34×	27.77	1.45×
16:32	136.89	1.42×	32.13	1.48×	53.73	1.35×	27.81	1.45×

4.5 Ablation Study

Ablation over calibration context source (Table 4). We examine whether calibration tokens must come from the model being pruned (self-generated) or can be produced by a different model (cross-model). Specifically, we compare calibrating the 14B model on its own tokens with using tokens generated by the larger 32B model when pruning the 14B model. Self-generated calibration performs better in both settings, achieving 4.18 vs. 4.13 under $2{:}4$ sparsity and 5.35 vs. 5.23 under 50% unstructured sparsity. This suggests that generating calibration tokens with the model being pruned is a contributing factor, and using sequences generated by a different model introduces a distributional discrepancy in the calibration data.

Table 4: **Results on the WritingBench benchmark under self-generated and cross-model calibration.** Qwen3-14B is pruned using activations from its own generations **(self-generated)** or from Qwen3-32B generations **(cross-model)**.

Model	Sparsity	Calibration	Overall
Qwen3-14B	50%	Cross-model (32B)	5.23
		Self-generated (14B)	**5.35**
	2:4	Cross-model (32B)	4.13
		Self-generated (14B)	**4.18**

Varying the pruning backend (Table 5). All preceding experiments use SparseGPT (Frantar & Alistarh, 2023) as the pruning backend when contrasting C4 with SparseDecoding, so the reported gains could in principle be specific to the solver. To verify the improvements are robust to the choice of pruning backend, we present the comparison with Wanda (Sun et al., 2024) in Table 5. SparseDecoding remains consistently ahead of C4 under both sparsity settings, indicating that improvements come from the calibration activations themselves rather than from the pruning backend.

Table 5: **Results on the WritingBench benchmark under 50% and 2:4 sparsity with Wanda on the Qwen3 model family.**

Model	Sparsity	Method	Overall
Qwen3-14B	50%	C4	4.94
		SparseDecoding	**5.23**
	2:4	C4	2.41
		SparseDecoding	**3.30**
Qwen3-32B	50%	C4	5.52
		SparseDecoding	**5.71**
	2:4	C4	3.34
		SparseDecoding	**4.39**

5 Conclusion

This work presents *SparseDecoding*, a decoding-aware pruning framework for LLMs that bridges the gap between pruning calibration objectives and downstream decoding behavior. By incorporating autoregressive outputs into calibration, SparseDecoding improves generation quality without additional training. Additionally, a N:M SpMV kernel is introduced to achieve practical speedup with the weight sparsity. Extensive experiments across multiple LLMs and sparsity patterns demonstrate that SparseDecoding consistently outperforms conventional calibration approaches, highlighting the importance of decoding-aware calibration for training-free LLM pruning. On top of the method, our N:M SpMV kernel achieves up to $1.48\times$ end-to-end decoding speedup on NVIDIA A100 GPUs.

AI USE STATEMENT

In this work, we used generative AI tools only to aid and polish the writing of this paper, including improving grammar and clarity, formatting references, and checking consistency between the text and tables. We did not use generative AI tools for other tasks, such as research ideation, experimental design, implementation, data generation, proofs, or interpretation of results. The authors reviewed and revised all AI-assisted edits. We take full responsibility for the final content of the paper.

REFERENCES

Yushi Bai, Jiajie Zhang, Xin Lv, Linzhi Zheng, Siqi Zhu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longwriter: Unleashing 10,000+ word generation from long context llms. In *ICLR*, 2025.

Abhinav Bandari, Lu Yin, Cheng-Yu Hsieh, Ajay Kumar Jaiswal, Tianlong Chen, Li Shen, Ranjay Krishna, and Shiwei Liu. Is c4 dataset optimal for pruning? an investigation of calibration data for LLM pruning. In *EMNLP*, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Xin Dong, Shangyu Chen, and Sinno Jialin Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *NeurIPS*, 2017.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*, 2023.

Gongfan Fang, Hongxu Yin, Saurav Muralidharan, Greg Heinrich, Jeff Pool, Jan Kautz, Pavlo Molchanov, and Xinchao Wang. MaskLLM: Learnable semi-structured sparsity for large language models. In *NeurIPS*, 2024.

Elias Frantar and Dan Alistarh. Optimal brain compression: A framework for accurate post-training quantization and pruning. In *NeurIPS*, 2022.

Elias Frantar and Dan Alistarh. SparseGPT: Massive language models can be accurately pruned in one-shot. In *ICML*, 2023.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Break the sequential dependency of llm inference using lookahead decoding. In *ICML*, 2024.

Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The Pile: An 800GB dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Aaron Grattafiori et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *NeurIPS*, 2015.

Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *ICLR*, 2016.

Babak Hassibi and David Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *NeurIPS*, 1992.

Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *ICCV*, 2017.

Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: pruning and growth for efficient inference and training in neural networks. *JMLR*, 22 (241):1–124, 2021.

Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. Flashdecoding++: Faster large language model inference on gpus. In *MLSys*, 2024.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *ICLR*, 2025.

Yixin Ji, Yang Xiang, Juntao Li, Qingrong Xia, Ping Li, Xinyu Duan, Zhefeng Wang, and Min Zhang. Beware of calibration data for pruning large language models. In *ICLR*, 2025.

Woojeong Kim, Junxiong Wang, Jing Nathan Yan, Mohamed Abdelfattah, and Alexander M. Rush. Overfill: Two-stage models for efficient language model decoding. In *COLM*, 2025.

Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. In *NeurIPS*, 1989.

Bin Lin, Ningxin Zheng, Lei Wang, Shijie Cao, Lingxiao Ma, Quanlu Zhang, Yi Zhu, Ting Cao, Jilong Xue, Yuqing Yang, and Fan Yang. Efficient gpu kernels for n:m-sparse weights in deep learning. In *MLSys*, 2023.

Ryan Lucas, Kayhan Behdin, Zhipeng Wang, Qingquan Song, Shao Tang, and Rahul Mazumder. Reasoning models can be accurately pruned via chain-of-thought reconstruction. In *ICLR*, 2026.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. Llm-pruner: On the structural pruning of large language models. In *NeurIPS*, 2023.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *ICLR*, 2017.

Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021.

NVIDIA Corporation. cuSPARSELt: A High-Performance CUDA Library for Sparse Matrix-Matrix Multiplication. *NVIDIA Technical Documentation*, 2021.

Jeff Pool and Chong Yu. Channel permutations for n:m sparsity. In *NeurIPS*, 2021.

PyTorch. Gpt-fast: Simple and efficient pytorch-native transformer text generation. *GitHub repository*, 2023.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR*, 2020.

Sidak Pal Singh and Dan Alistarh. Woodfisher: Efficient second-order approximation for neural network compression. In *NeurIPS*, 2020.

Mingluo Su and Huan Wang. Rose: Reordered sparsegpt for more accurate one-shot large language models pruning. In *Conference on Parsimony and Learning*, 2026.

Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. A simple and effective pruning approach for large language models. In *ICLR*, 2024.

Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *MAPL*, 2019.

Huan Wang, Yulun Zhang, Can Qin, Luc Van Gool, and Yun Fu. Global aligned structured sparsity learning for efficient image super-resolution. *TPAMI*, 45(9):10974–10989, 2023.

Ziyan Wang, Enmao Diao, Qi Le, Pu Wang, Guanchu Wang, Minwoo Lee, Shu ping Yeh, and Li Yang. Think before you prune: Self-reflective structured pruning for reasoning language models. *arXiv preprint arXiv:2512.02185*, 2025.

Maurice Weber, Daniel Fu, Quentin Anthony, Yonatan Oren, Shane Adams, Anton Alexandrov, Xiaozhong Lyu, Huu Nguyen, Xiaozhe Yao, Virginia Adams, Ben Athiwaratkun, Rahul Chalamala, Kezhen Chen, Max Ryabinin, Tri Dao, Percy Liang, Christopher Ré, Irina Rish, and Ce Zhang. Redpajama: an open dataset for training large language models, 2024.

Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *NeurIPS*, 2016.

Yuning Wu, Jiahao Mei, Ming Yan, Chenliang Li, Shaopeng Lai, Yuran Ren, Zijia Wang, Ji Zhang, Mengyue Wu, Qin Jin, and Fei Huang. Writingbench: A comprehensive benchmark for generative writing. In *NeurIPS*, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yuxin Zhang, Mingbao Lin, ZhiHang Lin, Yiting Luo, Ke Li, Fei Chao, Yongjian Wu, and Rongrong Ji. Learning best combination for efficient n:m sparsity. In *NeurIPS*, 2022.

A Analysis of Our Method

In this section, we theoretically characterize the discrepancy between a calibration objective and the decode-time reconstruction objective, and empirically compare this discrepancy for autoregressive and fixed-text calibration.

A.1 Theoretical Analysis

Setup. We follow the notation introduced in Sec. 3, and recall that $\Delta W_\ell := W_\ell - \widehat{W}_\ell$ denotes the weight difference between the dense weights and the sparsified weights of layer ℓ. For an activation source $R \in \{\mathrm{Dec}, \mathrm{AR}, \mathrm{TF}\}$, Dec denotes decode-step activations produced by the dense model during autoregressive generation on held-out target-task prompts, AR denotes decode-step activations produced by the dense model on the calibration prompts used by SparseDecoding, and TF denotes activations obtained from fixed calibration sequences. We let $x_\ell^R \in \mathbb{R}^{d_{\ell,\mathrm{in}}}$ denote a random input activation of layer ℓ drawn from source R, and define the associated population Hessian as

$$H_\ell^R := 2\,\mathbb{E}\left[x_\ell^R (x_\ell^R)^\top\right] \in \mathbb{R}^{d_{\ell,\mathrm{in}} \times d_{\ell,\mathrm{in}}} \tag{6}$$

Since H_ℓ^{Dec} is symmetric positive semidefinite, let $\{u_{\ell,i}\}_{i=1}^{d_{\ell,\mathrm{in}}}$ be an orthonormal eigenbasis satisfying

$$H_\ell^{\mathrm{Dec}} u_{\ell,i} = \lambda_{\ell,i} u_{\ell,i}, \qquad \lambda_{\ell,1} \geq \cdots \geq \lambda_{\ell,d_{\ell,\mathrm{in}}} \geq 0. \tag{7}$$

For any k such that $\lambda_{\ell,k} > 0$, we define

$$U_{\ell,k} := [u_{\ell,1}, \ldots, u_{\ell,k}], \quad \Lambda_{\ell,k} := \mathrm{diag}(\lambda_{\ell,1}, \ldots, \lambda_{\ell,k}), \quad P_{\ell,k} := U_{\ell,k} U_{\ell,k}^\top. \tag{8}$$

Here, $U_{\ell,k} \in \mathbb{R}^{d_{\ell,\mathrm{in}} \times k}$ contains the top-k eigenvectors of H_ℓ^{Dec}, $\Lambda_{\ell,k} \in \mathbb{R}^{k \times k}$ contains their corresponding eigenvalues, and $P_{\ell,k} \in \mathbb{R}^{d_{\ell,\mathrm{in}} \times d_{\ell,\mathrm{in}}}$ is the orthogonal projector onto the span of the eigenvectors.

Definition A.1. (Normalized decode-time calibration discrepancy). *For a calibration activation source $C \in \{\mathrm{AR}, \mathrm{TF}\}$, we define*

$$\widetilde{H}_{\ell,C}^{(k)} := \Lambda_{\ell,k}^{-1/2} U_{\ell,k}^\top H_\ell^C U_{\ell,k} \Lambda_{\ell,k}^{-1/2} \in \mathbb{R}^{k \times k}, \tag{9}$$

and

$$\epsilon_{\ell,C}^{(k)} := \left\| \widetilde{H}_{\ell,C}^{(k)} - I_k \right\|_{\mathrm{op}}. \tag{10}$$

where $\widetilde{H}_{\ell,C}^{(k)}$ is the calibration Hessian H_ℓ^C restricted to the top-k decode-time eigenvectors and rescaled by the corresponding decode-time eigenvalues. The scalar $\epsilon_{\ell,C}^{(k)}$ is the operator (spectral) norm of the difference between the normalized calibration Hessian and the identity matrix I_k.

Remark A.1. We remark that the identity matrix I_k is the normalized form of the decode-time Hessian, since

$$\Lambda_{\ell,k}^{-1/2} U_{\ell,k}^\top H_\ell^{\mathrm{Dec}} U_{\ell,k} \Lambda_{\ell,k}^{-1/2} = I_k. \tag{11}$$

Thus, $\epsilon_{\ell,C}^{(k)} = 0$ indicates that the calibration Hessian exactly matches the decode-time Hessian after projection onto the top-k decode-time eigenvectors and normalization by the corresponding decode-time eigenvalues. More generally, a smaller $\epsilon_{\ell,C}^{(k)}$ indicates a smaller worst-case relative discrepancy between the calibration and decode-time reconstruction objectives over these directions.

Definition A.2. (Projected reconstruction loss). *For $R \in \{\mathrm{Dec}, \mathrm{AR}, \mathrm{TF}\}$, we define the layer-wise reconstruction loss over the top-k eigenvectors of the decode-time Hessian as*

$$\begin{aligned} \mathcal{L}_{\ell,R}^{(k)}(\Delta W_\ell) &:= \mathbb{E}\left[\left\| \Delta W_\ell P_{\ell,k} x_\ell^R \right\|_2^2 \right] \\ &= \frac{1}{2} \mathrm{tr}\left(\Delta W_\ell P_{\ell,k} H_\ell^R P_{\ell,k} \Delta W_\ell^\top \right). \end{aligned} \tag{12}$$

Theorem A.1. (Calibration reconstruction-loss discrepancy). *For every $C \in \{\mathrm{AR}, \mathrm{TF}\}$,*

$$\epsilon_{\ell,C}^{(k)} = \max_{\Delta W_\ell : \mathcal{L}_{\ell,\mathrm{Dec}}^{(k)}(\Delta W_\ell) > 0} \frac{\left| \mathcal{L}_{\ell,C}^{(k)}(\Delta W_\ell) - \mathcal{L}_{\ell,\mathrm{Dec}}^{(k)}(\Delta W_\ell) \right|}{\mathcal{L}_{\ell,\mathrm{Dec}}^{(k)}(\Delta W_\ell)}. \tag{13}$$

Proof. We first fix $C \in \{\mathrm{AR}, \mathrm{TF}\}$, and then define $Z := \Delta W_\ell U_{\ell,k} \Lambda_{\ell,k}^{1/2}$ and $M_C := \widetilde{H}_{\ell,C}^{(k)} - I_k$ to simplify the notation in the remainder of the proof. We first begin our proof by subtracting the two losses

$$\begin{aligned} \mathcal{L}_{\ell,C}^{(k)}(\Delta W_\ell) - \mathcal{L}_{\ell,\mathrm{Dec}}^{(k)}(\Delta W_\ell) &= \frac{1}{2} \mathrm{tr}\left(Z \widetilde{H}_{\ell,C}^{(k)} Z^\top \right) - \frac{1}{2} \mathrm{tr}\left(Z Z^\top \right) && \text{(Definitions of two losses and } Z\text{)} \\ &= \frac{1}{2} \mathrm{tr}\left(Z \widetilde{H}_{\ell,C}^{(k)} Z^\top - Z Z^\top \right) && \text{(linearity of trace)} \qquad (14) \\ &= \frac{1}{2} \mathrm{tr}\left(Z \left(\widetilde{H}_{\ell,C}^{(k)} - I_k \right) Z^\top \right) \\ &= \frac{1}{2} \mathrm{tr}\left(Z M_C Z^\top \right) && \text{(Definition of } M_C\text{)} \qquad (15) \end{aligned}$$

Since M_C is symmetric,

$$\begin{aligned}
\left|\mathcal{L}^{(k)}_{\ell,C}(\Delta W_\ell) - \mathcal{L}^{(k)}_{\ell,\mathrm{Dec}}(\Delta W_\ell)\right| &= \frac{1}{2}\left|\mathrm{tr}(ZM_CZ^\top)\right| && \text{(Eq. 15)} \\
&= \frac{1}{2}\left|\langle Z, ZM_C\rangle_F\right| && \text{(Definition of } \langle\cdot,\cdot\rangle_F) \quad (16) \\
&\le \frac{1}{2}\|Z\|_F\,\|ZM_C\|_F && \text{(Cauchy-Schwarz inequality)} \quad (17) \\
&\le \frac{1}{2}\|Z\|_F^2\,\|M_C\|_{\mathrm{op}} \\
&= \epsilon^{(k)}_{\ell,C}\mathcal{L}^{(k)}_{\ell,\mathrm{Dec}}(\Delta W_\ell). && \text{(Definitions of } Z, M_C, \text{Eqs. 10, 12)} \\
& && (18)
\end{aligned}$$

where $\langle\cdot,\cdot\rangle_F$ denotes the Frobenius inner product, which proves the upper bound in Eq. (13) by dividing by $\mathcal{L}^{(k)}_{\ell,\mathrm{Dec}}(\Delta W_\ell)$.

To show that the bound in Eq. (18) is tight, we construct a specific ΔW_ℓ that attains the upper bound. Since M_C is symmetric, it admits an eigendecomposition with real eigenvalues and orthonormal eigenvectors. We let $v \in \mathbb{R}^k$ be an eigenvector of M_C whose eigenvalue λ^* has the largest absolute value so that $|\lambda^*| = \|M_C\|_{\mathrm{op}}$. Then we let $Z^* \in \mathbb{R}^{d_{\ell,\mathrm{out}}\times k}$ be the rank-one matrix in which the first row is set to $v^\top$, while the remaining rows are zero. Then the following conditions hold:

$$\|Z^*\|_F^2 = \|v\|_2^2 = 1, \quad \left|\mathrm{tr}(Z^*M_CZ^{*\top})\right| = \left|v^\top M_C v\right| = |\lambda^*| = \|M_C\|_{\mathrm{op}} \tag{19}$$

As the top-k eigenvectors are orthonormal, then the conditions $\lambda_{\ell,k} > 0$ and $U_{\ell,k}^\top U_{\ell,k} = I_k$ hold, we therefore define $\Delta W_\ell^* := Z^*\Lambda_{\ell,k}^{-1/2}U_{\ell,k}^\top$ which recovers Z^* under the substitution used throughout the proof by $\Delta W_\ell^* U_{\ell,k}\Lambda_{\ell,k}^{1/2} = Z^*\Lambda_{\ell,k}^{-1/2}U_{\ell,k}^\top U_{\ell,k}\Lambda_{\ell,k}^{1/2} = Z^*$. In particular, we can derive the reconstruction objective

$$\mathcal{L}^{(k)}_{\ell,\mathrm{Dec}}(\Delta W_\ell^*) = \frac{1}{2}\|Z^*\|_F^2 = \frac{1}{2} > 0,$$

hence ΔW_ℓ^* belongs to the feasible set of the maximization in Eq. (13). Substituting into Eq. (19) gives

$$\frac{\left|\mathcal{L}^{(k)}_{\ell,C}(\Delta W_\ell^*) - \mathcal{L}^{(k)}_{\ell,\mathrm{Dec}}(\Delta W_\ell^*)\right|}{\mathcal{L}^{(k)}_{\ell,\mathrm{Dec}}(\Delta W_\ell^*)} = \frac{\frac{1}{2}\left|\mathrm{tr}(Z^*M_CZ^{*\top})\right|}{\frac{1}{2}\|Z^*\|_F^2} = \|M_C\|_{\mathrm{op}} = \epsilon^{(k)}_{\ell,C},$$

so the upper bound is achieved exactly at ΔW_ℓ^*, completing the proof of Eq. (13). □

Remark A.2. We remark that by Theorem A.1, for a layer ℓ and retained dimension k, if

$$\epsilon^{(k)}_{\ell,\mathrm{AR}} < \epsilon^{(k)}_{\ell,\mathrm{TF}},$$

then AR calibration has a smaller worst-case relative reconstruction-loss discrepancy with respect to the decode-time objective than TF calibration at layer ℓ.

A.2 Empirical Findings

In this section, we use prompts from LongWriter (Bai et al., 2025) to generate the autoregressive calibration activations. We instantiate the fixed calibration source TF using the C4 dataset (Raffel et al., 2020), hence, we denote the fixed calibration source as C4 within this section. We compute the finite-sample version of the calibration discrepancy in Eq. (10) for Qwen3-14B and 32B (Yang et al., 2025). For each retained fraction $\rho \in \{0.01, 0.02, 0.05, 0.10, 0.20, 0.50\}$, we set $k_\ell(\rho) = \max\{1, \lceil \rho d_{\ell,\mathrm{in}}\rceil\}$ as the number of top-k eigenvectors to retain. Let $\widehat{H}_\ell^C$ and $\widehat{H}_\ell^{\mathrm{Dec}}$ denote the empirical Hessians constructed from collected activation matrices. More generally, for $R \in \{\mathrm{Dec}, \mathrm{AR}, \mathrm{TF}\}$, we let $X_\ell^R = [x_{\ell,1}^R, \ldots, x_{\ell,N_R}^R] \in \mathbb{R}^{d_{\ell,\mathrm{in}}\times N_R}$ denote the collected activation matrix where N_R denotes the number of collected activation samples from the source R. We therefore estimate the corresponding population Hessian H_ℓ^R by

$$\widehat{H}_\ell^R := \frac{2}{N_R}X_\ell^R(X_\ell^R)^\top = \frac{2}{N_R}\sum_{i=1}^{N_R} x_{\ell,i}^R(x_{\ell,i}^R)^\top. \tag{20}$$

Table 6: **Empirical comparison of AR and C4 calibration discrepancy on Qwen3 14B.** For each retained eigenvector fraction, we report the number and percentage of prunable modules (out of 280) where $\hat{\epsilon}^{(k)}_{\ell,\mathrm{AR}} < \hat{\epsilon}^{(k)}_{\ell,\mathrm{C4}}$, together with the median of the discrepancy ratio $\hat{\epsilon}_{\mathrm{C4}}/\hat{\epsilon}_{\mathrm{AR}}$.

Retained eigenvectors	Modules with $\hat{\epsilon}_{\mathrm{AR}} < \hat{\epsilon}_{\mathrm{C4}}$	Percentage	Median ratio $\hat{\epsilon}_{\mathrm{C4}}/\hat{\epsilon}_{\mathrm{AR}}$
1%	274	97.9%	2.30
2%	271	96.8%	2.39
5%	266	95.0%	2.65
10%	264	94.3%	2.83
20%	261	93.2%	3.06
50%	248	88.6%	3.26

Table 7: **Empirical comparison of AR and C4 calibration discrepancy on Qwen3 32B.** For each retained eigenvector fraction, we report the number and percentage of prunable modules (out of 448) where $\hat{\epsilon}^{(k)}_{\ell,\mathrm{AR}} < \hat{\epsilon}^{(k)}_{\ell,\mathrm{C4}}$, together with the median of the discrepancy ratio $\hat{\epsilon}_{\mathrm{C4}}/\hat{\epsilon}_{\mathrm{AR}}$.

Retained eigenvectors	Modules with $\hat{\epsilon}_{\mathrm{AR}} < \hat{\epsilon}_{\mathrm{C4}}$	Percentage	Median ratio $\hat{\epsilon}_{\mathrm{C4}}/\hat{\epsilon}_{\mathrm{AR}}$
1%	390	87.1%	1.81
2%	419	93.5%	1.92
5%	440	98.2%	2.08
10%	439	98.0%	2.16
20%	444	99.1%	2.17
50%	442	98.7%	2.19

We therefore obtain $\widehat{U}_{\ell,k}$ and $\widehat{\Lambda}_{\ell,k}$ from $\widehat{H}^{\mathrm{Dec}}_{\ell}$ by Eq. (8), and define

$$\hat{\epsilon}^{(k)}_{\ell,C} := \left\| \widehat{\Lambda}^{-1/2}_{\ell,k}\, \widehat{U}^{\top}_{\ell,k}\, \widehat{H}^{C}_{\ell}\, \widehat{U}_{\ell,k}\, \widehat{\Lambda}^{-1/2}_{\ell,k} - I_k \right\|_{\mathrm{op}} \tag{21}$$

as the finite-sample estimate of $\epsilon^{(k)}_{\ell,C}$ in Eq. (10), obtained by substituting empirical Hessians for the population Hessians. By Theorem A.1, its population counterpart $\epsilon^{(k)}_{\ell,C}$ equals the worst-case relative reconstruction loss discrepancy. We therefore use $\hat{\epsilon}^{(k)}_{\ell,C}$ as an empirical estimate of this quantity. By Remark A.2, a smaller value indicates a smaller empirical worst-case reconstruction-loss discrepancy. The median ratio of the estimates is reported as $\hat{\epsilon}_{\mathrm{C4}}/\hat{\epsilon}_{\mathrm{AR}}$, where $\hat{\epsilon}_{\mathrm{C4}}/\hat{\epsilon}_{\mathrm{AR}} > 1$ indicates that AR has a smaller calibration discrepancy than C4. Table 6 and 7 report the empirical statistics of Qwen3-14B and 32B, respectively. Across both models and all retained fractions, AR calibration yields a smaller discrepancy in at least 87% of the prunable modules, with median ratios $\hat{\epsilon}_{\mathrm{C4}}/\hat{\epsilon}_{\mathrm{AR}}$ between 1.81 and 3.26, indicating that decode-time calibration consistently aligns more closely with the decode-time reconstruction objective than fixed-text calibration.

B Additional Experimental Results

SparseDecoding for 8:16 sparsity (Table 8). We additionally evaluate SparseDecoding under the $8{:}16$ sparsity setting. Compared with C4 calibration, SparseDecoding consistently improves generation quality across the evaluated models. These results show that the benefit of decoding-aware calibration is not limited to $2{:}4$ sparsity, but also extends to the $8{:}16$ pattern.

Ablation over calibration datasets (Table 9). We extend our comparison to two additional widely used calibration datasets, the Pile (Gao et al., 2020) and RedPajama (Weber et al., 2024), evaluating Qwen3-14B and Qwen3-32B under 50% unstructured and 2:4 semi-structured sparsity. SparseDecoding consistently achieves higher overall WritingBench scores than calibration with either corpus across both models and sparsity settings.

Table 8: **Results on WritingBench for Qwen3-14B and Qwen3-32B under unstructured** 50% **and semi-structured** 8:16 **sparsity. SparseDecoding (ours)** calibrates on autoregressive generations conditioned on LongWriter prompts; C4 uses fixed C4 sequences. Dense is the unpruned reference. "C" indicates the corresponding category-specific score. Bold marks the best overall score within each sparsity setting.

Model	Sparsity	Method	Overall	Domains						Requirements					
				D1	D2	D3	D4	D5	D6	R1	C	R2	C	R3	C
Qwen3-14B	0%	Dense	5.97	6.0	5.8	6.1	5.7	6.3	6.1	6.1	6.4	6.0	6.8	6.0	6.3
	50%	C4	4.62	4.6	4.4	4.6	4.0	5.4	5.2	4.8	5.0	4.6	5.2	4.5	4.6
		SparseDecoding	**5.35**	5.3	5.3	5.4	4.9	5.9	5.6	5.5	5.8	5.4	6.2	5.4	5.6
	8:16	C4	4.50	4.5	4.4	4.5	3.8	5.0	5.2	4.6	4.8	4.5	5.0	4.3	4.2
		SparseDecoding	**5.31**	5.2	5.3	5.3	4.7	5.9	5.7	5.4	5.6	5.4	6.1	5.2	5.4
Qwen3-32B	0%	Dense	6.48	6.5	6.4	6.5	6.3	6.7	6.6	6.6	6.8	6.5	7.2	6.6	6.8
	50%	C4	5.34	5.2	5.2	5.4	4.9	5.9	5.8	5.5	5.7	5.3	6.0	5.3	5.4
		SparseDecoding	**6.09**	6.0	6.0	6.1	5.8	6.5	6.3	6.2	6.5	6.1	6.8	6.1	6.4
	8:16	C4	5.20	5.2	5.1	5.3	4.5	5.6	5.7	5.3	5.5	5.2	5.8	5.1	5.2
		SparseDecoding	**5.91**	5.8	5.9	5.9	5.5	6.3	6.3	6.0	6.3	5.9	6.7	6.0	6.1

Table 9: **Results on WritingBench benchmark under** 50% **and** 2:4 **sparsity with different calibration data for the Qwen3 model family.** Pile and RedPajama use fixed pretraining-corpus sequences; **SparseDecoding** calibrates on autoregressive generations conditioned on LongWriter prompts.

Model	Sparsity	Calibration	Overall	Domains						Requirements					
				D1	D2	D3	D4	D5	D6	R1	C	R2	C	R3	C
Qwen3-14B	0%	Dense	5.97	6.0	5.8	6.1	5.7	6.3	6.1	6.1	6.4	6.0	6.8	6.0	6.3
	50%	Pile	5.00	5.2	5.1	5.1	4.0	5.7	5.3	5.0	5.2	5.0	5.6	4.8	5.0
		RedPajama	5.04	5.0	5.0	5.1	4.4	5.6	5.6	5.2	5.4	5.1	5.7	4.9	5.0
		SparseDecoding	**5.35**	5.3	5.3	5.4	4.9	5.9	5.6	5.5	5.8	5.4	6.2	5.4	5.6
	2:4	Pile	2.68	3.0	2.7	2.6	1.9	3.0	3.1	2.7	2.7	2.7	2.9	2.4	2.2
		RedPajama	2.62	2.8	2.6	2.5	2.1	2.7	3.3	2.7	2.7	2.6	2.7	2.4	2.2
		SparseDecoding	**4.18**	4.2	4.3	4.3	3.3	4.8	4.6	4.2	4.4	4.2	4.9	4.1	4.2
Qwen3-32B	0%	Dense	6.48	6.5	6.4	6.5	6.3	6.7	6.6	6.6	6.8	6.5	7.2	6.6	6.8
	50%	Pile	5.80	5.8	5.8	5.8	5.3	6.2	6.2	5.9	6.1	5.8	6.5	5.8	6.0
		RedPajama	5.72	5.7	5.7	5.8	5.3	6.1	6.1	5.8	6.1	5.7	6.4	5.7	5.7
		SparseDecoding	**6.09**	6.0	6.0	6.1	5.8	6.5	6.3	6.2	6.5	6.1	6.8	6.1	6.4
	2:4	Pile	3.32	3.6	3.3	3.3	2.6	3.6	3.9	3.4	3.4	3.3	3.6	3.2	3.2
		RedPajama	3.74	3.8	3.7	3.7	3.0	4.2	4.5	3.8	3.9	3.8	4.3	3.7	3.5
		SparseDecoding	**5.31**	5.2	5.3	5.4	4.7	5.8	5.7	5.4	5.6	5.3	6.0	5.3	5.5

C KERNEL IMPLEMENTATION DETAILS

Implementation. Our $N{:}M$ SpMV kernel is implemented in Triton (Tillet et al., 2019) and optimized for batch-one autoregressive decoding. We avoid explicitly staging the input vector in shared memory, where input-vector loads use the `.ca` cache policy[1], so that activations reused across output rows can benefit from both L1 and L2 caching, while sparse weights and bitmask metadata use the `.cg` policy and are streamed primarily through L2 since they are primarily streamed and have limited temporal reuse. We compile a separate kernel for each $N{:}M$ pattern, fixing N and M at compile time. This fixes the number of nonzeros per group and the bitmask-decoding loop bounds at compile time, allowing the compiler to fully unroll the corresponding loops. Each nonzero position is recovered using a bit-scan operation that locates the least significant set bit. We also autotune the output tile size, number of warps, pipeline stages, and reduction strategy for each matrix shape and

[1] See the NVIDIA PTX ISA documentation for definitions of cache operators.

sparsity pattern. For configurations that benefit from additional parallelism, we split the reduction across multiple programs and combine the partial sums with `atomicAdd`.