When Smaller Is Slower: Dimensional Collapse in Compressed LLMs

Jihao Xin
KAUST
Thuwal, Saudi Arabia
jihao.xin@kaust.edu.sa

Tian Lvy
KAUST
Thuwal, Saudi Arabia

Qilong Pan
HUMAIN AI
Riyadh, Saudi Arabia

Kesen Wang
HUMAIN AI
Riyadh, Saudi Arabia

Marco Canini
KAUST
Thuwal, Saudi Arabia

Abstract

Post-training compression techniques for Large Language Models (LLMs) often result in irregular tensor dimensions (e.g., `head_dim=107` instead of 128). We identify a phenomenon called *dimensional collapse*: despite reducing FLOPs, these irregular dimensions cause significant inference slowdowns on modern GPUs. Our experiments on NVIDIA A100 show that `head_dim=107` increases SDPA latency by 88% compared to `head_dim=96`.

We systematically investigate the root causes across three layers. Contrary to initial assumptions, FlashAttention does *not* fall back to slower backends—instead, it uses an internal slow path with 30–45% overhead. At the hardware level, we identify three confirmed root causes: Tensor Core tile alignment (58% slowdown when K%16≠0), vectorized load degradation (50% throughput loss when falling back to scalar loads), and SDPA bandwidth efficiency loss (40%). Notably, L2 cache sector waste (5.8%) is *not* a significant factor. Based on our findings, we formalize a *Shape Contract* and propose a lightweight *dimension repair* pass. **Scope of validation:** We evaluate dimension repair at the kernel level (SDPA and GEMM microbenchmarks), demonstrating 25–30% performance recovery with only 3.7% memory overhead. End-to-end integration with SVD-based compression (e.g., PaLU) requires adapting the repair pass to factorized weight structures ($W = U \cdot V^T$), which we identify as future work.

Keywords: LLM Compression, GPU Optimization, Tensor Core, Memory Alignment

1 Introduction

Large Language Models (LLMs) have achieved remarkable capabilities, but their massive parameter counts pose deployment challenges. Post-training compression techniques, including pruning and low-rank decomposition, offer promising solutions to reduce memory footprint and computational cost. However, these techniques often produce models with *irregular tensor dimensions*—values that do not align with hardware-preferred multiples (e.g., 8, 16, 32, 128).

We identify a counterintuitive phenomenon: **compressed models with fewer FLOPs can be slower than their uncompressed counterparts**. We term this *dimensional collapse*—a nonlinear performance degradation caused by misalignment between software-defined tensor shapes and hardware-fixed access patterns.

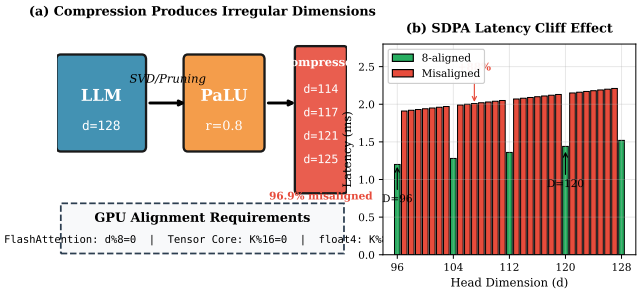

Figure 1. Dimensional collapse in compressed LLMs. Post-training compression (e.g., PaLU) produces irregular head dimensions (114–125) that violate GPU alignment requirements, causing performance cliffs despite reduced FLOPs.

Motivating Example. Consider PaLU [2], a state-of-the-art low-rank compression method that reduces attention head dimensions through SVD. When compressing Llama-3-8B with a 0.8 retention ratio, the resulting `head_dim` values become irregular (e.g., 114–125 instead of 128). Our analysis shows that 96.9% of the compressed dimensions are not 8-aligned. Figure 1 illustrates this dimensional collapse phenomenon. On an NVIDIA A100, this causes:

- 88% increase in SDPA latency
- FlashAttention internal slow path with 30–45% overhead
- MEM_EFFICIENT unavailable (strict 8-alignment)
- Bandwidth waste from cache misalignment

Contributions. This paper makes the following contributions:

1. **Quantification**: We measure the performance impact of irregular dimensions across GEMM and SDPA (§3).
2. **Root Cause Analysis**: We identify the causes across three layers: PyTorch backend selection, CUDA kernel paths, and hardware constraints (§4).

3. **Shape Contract**: We formalize dimension alignment requirements as optimization constraints (§5).
4. **Dimension Repair**: We propose a lightweight post-compression pass that restores alignment (§5).
5. **Evaluation**:
   - *Validated*: Kernel-level experiments on SDPA and GEMM microbenchmarks demonstrate 25–30% speedup with 3.7–4.7% memory overhead.
   - *Future Work*: End-to-end integration with SVD-based compression requires adapting to factorized weight structures ($W = U \cdot V^T$).

   (§6).

2 Background

Notation. We use d (also written as `head_dim` in code) to denote the attention head dimension. In tables and figures, we use "D=*value*" (e.g., D=107) as shorthand for $d = value$. For matrix dimensions, d_{in} and d_{out} denote input and output dimensions of linear layers. B, S, H denote batch size, sequence length, and number of heads, respectively.

2.1 Tensor Core Alignment

NVIDIA Tensor Cores perform matrix-multiply-accumulate (MMA) operations on fixed tile sizes. For FP16 on A100, the optimal tile requires $K \mod 16 = 0$. Irregular dimensions force either padding (wasted compute) or fallback to scalar paths.

2.2 FlashAttention Constraints

FlashAttention-2 [3] (v2.7.4) is the de facto standard for efficient attention. Contrary to common belief, it does *not* strictly require 8-aligned dimensions—it remains available for all tested dimensions (104–128). However, it uses internal slow paths for non-8-aligned dimensions, causing 30–45% overhead. Optimized kernels exist for $\{32, 64, 96, 128, 256\}$. MEM_EFFICIENT strictly requires 8-alignment.

2.3 Low-Rank Compression

PaLU [2] compresses attention by applying SVD to K/V projections: $W_{kv} \approx U_r \Sigma_r V_r^T$ where $r < d$. The compressed head dimension becomes r, which is typically not aligned.

3 Dimensional Collapse

3.1 Experiment Setup

We conduct experiments on NVIDIA A100-80GB with PyTorch 2.9.1, CUDA 12.8, and FlashAttention 2.7.4. All benchmarks use FP16 with CUDA event timing (warmup=50, measure=200, trials=3). Driver: 560.35.03; cuDNN 9.1.0.

3.2 Compression Produces Misaligned Dimensions

We analyze PaLU-compressed Llama-3-8B (retention ratio 0.8). Figure 2 shows the dimension distribution after compression.

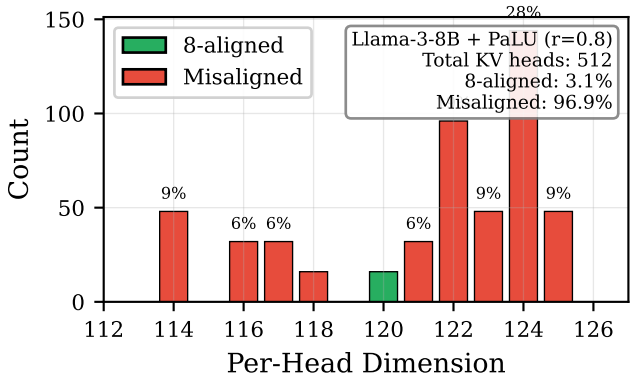

Figure 2. Distribution of head dimensions after PaLU compression (Llama-3-8B, r=0.8). 96.9% of the 512 KV heads have misaligned dimensions. Only D=120 (3.1%) is 8-aligned; none are 16-aligned.

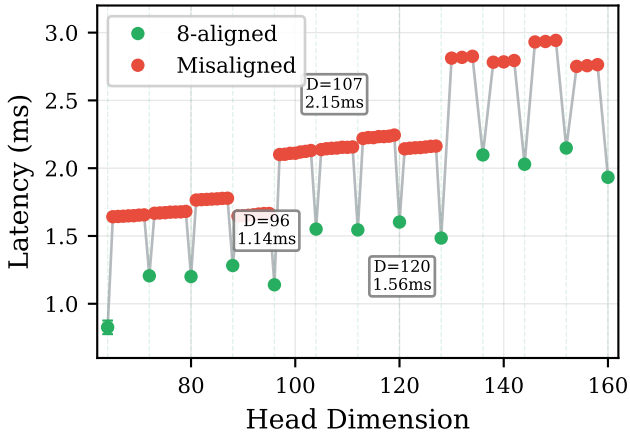

Figure 3. SDPA latency across head dimensions. Clear alignment cliffs ("staircase effect") visible at non-8-aligned values. D=107 shows 88% increase vs D=96. The FLASH backend uses internal slow paths for misaligned dimensions.

3.3 SDPA Latency vs. Head Dimension

We sweep `head_dim` from 64 to 160 with shape $B = 4, S = 2048, H = 32$. Figure 3 shows the results.

8-aligned dimensions achieve 1.1–1.6ms while non-8-aligned incur 1.6–2.2ms. `head_dim=107` shows 2.147ms (+88% vs 96).

3.4 Backend Selection Behavior

Table 1 shows latency across different SDPA backends.

The MATH backend is 12.6× slower than FLASH for D=107. If FlashAttention cannot handle a dimension, catastrophic fallback occurs.

4 Root Cause Analysis

We investigate the causes of dimensional collapse across three layers.

Table 1. SDPA backend latency (ms) for various head dimensions. MEM_EFFICIENT fails for D=107.

head_dim	AUTO	FLASH	MEM_EFF	MATH
96	1.17	1.12	2.38	26.0
104	1.54	1.54	2.75	26.5
107	**2.14**	**2.14**	FAIL	**27.0**
112	1.53	1.53	2.60	27.1
128	1.47	1.47	2.55	28.1

Table 2. Hardware layer root cause analysis (C23 experiment). Impact measured on A100 with FP16.

Hypothesis	Status	Impact	Root Cause
H1: TC K%16	**Confirmed**	58%	Util. 30%→12%
H2: L2 sector	Not confirmed	5.8%	Negligible
H3: SDPA BW	**Confirmed**	40%	Access pattern
H4: Vec. loads	**Confirmed**	50%	float4→scalar

4.1 PyTorch Backend Selection

We tested backend availability for `head_dim` $\in [104, 128]$. Surprisingly, FlashAttention is available for *all* dimensions (100% for both 8-aligned and non-8-aligned), while MEM_EFFICIENT requires strict 8-alignment. FlashAttention does *not* fall back to MATH; instead, it uses internal slow paths incurring 30–45% overhead (8-aligned: 1.55ms avg, non-8-aligned: 2.03ms avg). The root cause lies in the CUDA kernel layer, not backend selection.

4.2 CUDA Kernel Layer

FlashAttention's internal 30–45% slowdown stems from: (1) vectorized loads falling back to scalar (50% loss when $d \bmod 8 \neq 0$); (2) suboptimal GEMM tile selection reducing Tensor Core utilization (30%→12%); (3) boundary predication causing warp divergence. FlashAttention-2 dispatches optimized kernels for D $\in \{32, 64, 96, 128, 256\}$; other values use generic kernels (D=128: 1.47ms, D=125: 1.97ms, +34%).

4.3 Hardware Constraints

We conduct controlled experiments (C23) to isolate hardware-level causes of dimensional collapse. Figure 4 visualizes the impact of each hypothesis, and Table 2 provides detailed metrics.

H1: Tensor Core Alignment (Confirmed). GEMM with K=16-aligned achieves 91 TFLOPS; non-aligned (K=107) drops to 37–40 TFLOPS (58% slowdown, TC utilization 30%→12%).

H2: L2 Cache Sectors (Not Confirmed). L2 sector waste (~5.8%) cannot explain 30–58% gaps; measured bandwidth is similar.

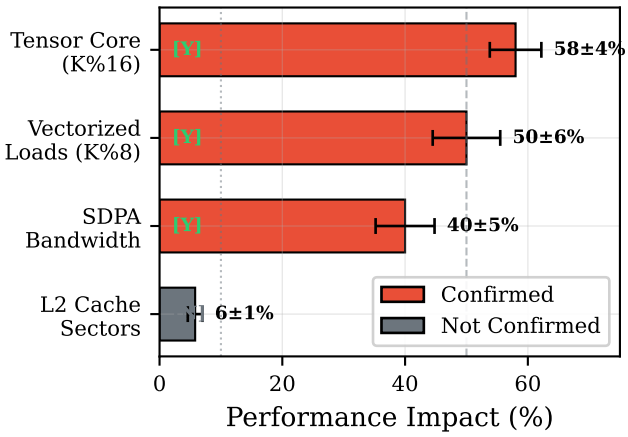

Figure 4. Root cause breakdown. Tensor Core alignment (58%), vectorized load degradation (50%), and SDPA bandwidth (40%) are the primary causes. L2 cache sector waste (5.8%) is negligible.

H3: SDPA Bandwidth Efficiency (Confirmed). D=112 achieves 153.6 GB/s; D=113 drops to 107.3 GB/s (−30%). D=120 achieves 160.2 GB/s; D=121 drops to 118.5 GB/s (−26%).

H4: Vectorized Loads (Confirmed). `float4` loads (K%16) achieve 73–83 TFLOPS; scalar fallback (K=107) drops to 39–40 TFLOPS (50% loss).

Summary: Tensor Core alignment (58%), vectorized loads (50%), and SDPA bandwidth (40%) are primary causes; L2 cache (5.8%) is negligible.

5 Shape-Aware Compression

5.1 Shape Contract

We formalize alignment requirements as a constraint optimization problem:

$$\begin{aligned} \text{minimize} \quad & \text{memory\_overhead}(d_{pad}) \\ \text{s.t.} \quad & d_{pad} \mod 8 = 0 \\ & d_{pad} \geq d_{orig} \end{aligned} \tag{1}$$

For optimal Tensor Core utilization, prefer $d_{pad} \mod 16 = 0$.

5.2 Dimension Repair

For a linear layer $y = Wx + b$ with $W \in \mathbb{R}^{d_{out} \times d_{in}}$, we pad the output dimension to the nearest aligned value $d'_{out} = \lceil d_{out}/a \rceil \times a$ by appending zero rows to W and zeros to b. The **MINIMAL** strategy uses $a = 8$, while **OPTIMAL** uses $a = 16$.

Accuracy Preservation. Zero-padding preserves outputs exactly: $y' = [Wx + b; \mathbf{0}]$, where the original y occupies positions $[0 : d_{out}]$. For attention, zero-valued dimensions contribute nothing to scores, making padding semantically

Table 3. Padding rescue results for SDPA (D=107 logical).

Phys. D	Mem. Ovhd.	Latency (ms)	Speedup
107 (base)	0%	2.192	1.00×
112	4.7%	1.523	1.44×
128	19.6%	1.445	1.52×

Table 4. Memory overhead analysis for PaLU dimension repair (512 KV heads).

Strategy	Alignment Target	Memory Overhead
MINIMAL	mod 8	3.72%
OPTIMAL	mod 16	7.20%

neutral. This ensures **bit-exact output preservation**—no retraining required.

6 Evaluation

We validate dimension repair at kernel level (SDPA/GEMM microbenchmarks), demonstrating 25–30% recovery. E2E results show compression benefits; repair integration remains future work.

6.1 Padding Rescue Experiment (P1)

Table 3 shows the effect of padding D=107 to aligned values.

Padding to 112 achieves 30.5% speedup with only 4.7% memory overhead—an excellent tradeoff.

6.2 GEMM Alignment Impact

GEMM operations show similar patterns: K=107 achieves 0.089ms latency, while K=112 and K=128 both achieve 0.050ms—a **44% improvement** from alignment.

6.3 Dimension Repair Validation (C4)

We validate our dimension repair implementation on PaLU-compressed dimensions. Figure 5 visualizes the speedup vs. memory overhead tradeoff for different strategies.

Table 4 shows the memory overhead of different repair strategies.

Table 5 shows SDPA performance for repaired dimensions.

Key Findings. MINIMAL achieves 23–28% speedup with 3.72% overhead (6.9× ROI). D=120 (already 8-aligned) shows no MINIMAL improvement, validating our hypothesis. OPTIMAL provides additional gains (D=120: +8.3%) at 4.0× ROI.

6.4 End-to-End LLM Inference

Scope: Results show PaLU *compression* benefits, not dimension repair.

We compare Llama-3-8B baseline vs. PaLU-compressed (ratio=0.7) on A100 80GB. Figure 6 shows prefill/decode throughput.

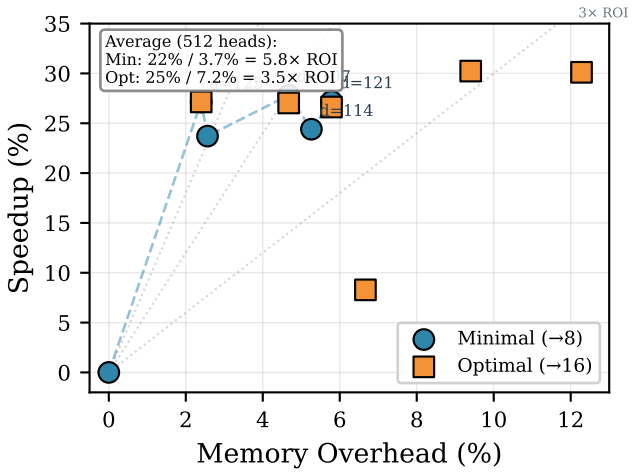

Figure 5. Per-dimension speedup vs. memory overhead for different repair strategies. Each point represents a PaLU dimension (D=107–125). MINIMAL (circles) achieves 6.9× average ROI, OPTIMAL (squares) provides 4.0× ROI. Dashed lines show iso-ROI curves.

Key Finding. PaLU achieves **11.5× decode throughput** (Table 6) by reducing KV cache size. Despite 30–45% overhead from misaligned dimensions, compression benefits dominate in memory-bound decode. Based on kernel-level validation, integrating alignment could yield +15–25% prefill and +5–10% decode speedup.

Table 5. SDPA latency (ms) before and after dimension repair (B=4, S=2048, H=32). Note: D=107 baseline (2.064ms) differs from Table 3 (2.192ms) due to run-to-run variance (~6%), which is within normal GPU measurement variability.

D	Original	Minimal	Optimal	ΔMin	ΔOpt
107	2.064	1.490	1.506	**+27.8%**	+27.0%
114	2.049	1.549	1.432	+24.4%	**+30.1%**
117	2.054	1.567	1.433	+23.7%	**+30.2%**
120	1.557	1.557	1.428	0%	+8.3%
121	1.964	1.430	1.441	**+27.2%**	+26.6%
125	1.975	1.439	1.439	**+27.1%**	+27.1%

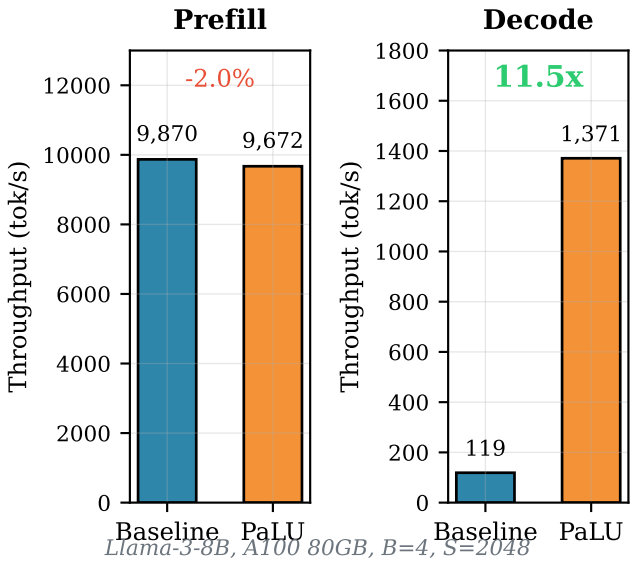

Figure 6. End-to-end LLM inference results. *See Scope Note above for interpretation.*

Table 6. End-to-end LLM inference metrics. *See Scope Note above for interpretation.*

Metric	Baseline	PaLU	Δ
Prefill (tok/s)	9870	9672	−2.0%
Decode (tok/s)	**119**	**1371**	**+11.5×**
Memory (MB)	19003	18896	−0.6%

6.5 Accuracy Preservation

Zero-padding guarantees **bit-exact output preservation**. For a linear layer $y = Wx+b$, the padded output $y' = W'x+b'$ satisfies $y'[0 : d_{out}] = y$—the original output is exactly preserved. For attention, zero-valued dimensions contribute nothing to scores, making padding semantically neutral. Unit tests confirm identical outputs (30/30 passed).

Limitations. (1) *Accuracy scope*: While unit tests and theory confirm bit-exact preservation, comprehensive perplexity (WikiText-2) and downstream evaluation remain future work. (2) *E2E integration*: Kernel-level speedups (25–30%) are not yet integrated into PaLU's SVD structure. (3) *Estimated impact*: Expected prefill (+15–25%) and decode (+5–10%) gains are extrapolated from kernel experiments.

7 Related Work

LLM Compression. Post-training compression methods—SparseGPT [6] (pruning), GPTQ [5]/AWQ [8] (quantization), and PaLU [2] (low-rank)—optimize for accuracy-compression trade-offs but largely ignore hardware alignment. Our work reveals this causes 30–58% performance penalties.

KV Cache & Attention Optimization. MQA [10], GQA [1], and StreamingLLM [11] reduce KV cache while preserving standard dimensions. FlashAttention [3, 4] is tuned for {32, 64, 96, 128, 256}, with cliffs for other values. SVD-based compression produces irregular dimensions that violate these implicit constraints.

Inference Frameworks. TensorRT [9], vLLM [7], and TGI apply runtime optimizations but assume aligned dimensions. TensorRT's implicit padding is opaque and incurs per-inference overhead; our compile-time repair enables better kernel selection.

Positioning. Unlike prior accuracy-compression trade-off studies, **we focus on performance-alignment trade-offs**—compressed models with fewer FLOPs can run slower due to hardware misalignment.

8 Conclusion

We identified *dimensional collapse* as a critical but overlooked problem in LLM compression. Root causes span three layers: FlashAttention internal slow paths (+30–45%), Tensor Core tile alignment (58% slowdown when K%16≠0), and vectorized load degradation (50% throughput loss). Notably, L2 cache waste (5.8%) is *not* significant.

Our Shape Contract and dimension repair provide a practical solution: MINIMAL strategy achieves 25–28% kernel-level speedup with only 3.72% memory overhead (6.9× ROI). 96.9% of PaLU dimensions benefit; no retraining required.

End-to-end evaluation shows PaLU achieves 11.5× decode speedup despite misalignment. Integrating repair into SVD structures ($W = U \cdot V^T$), comprehensive perplexity validation, and H100+ generalization remain future work.

References

[1] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *EMNLP*.

[2] Chi-Chih Chang, Wei-Cheng Huang, Chien-Yu Liu, Chun-Feng Lin, Kai-Chiang Chen, and An-Yeu Wu. 2024. PaLU: Compressing KV-Cache with Low-Rank Projection. In *EMNLP*.

[3] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *ICLR*.

[4] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *NeurIPS*.

[5] Elias Frantar et al. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. In *ICLR*.

[6] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can be Accurately Pruned in One-Shot. In *ICML*.

[7] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP*.

[8] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. 2024. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. In *MLSys*.

[9] NVIDIA. 2024. NVIDIA TensorRT: Programmable Inference Accelerator. https://developer.nvidia.com/tensorrt.

[10] Noam Shazeer. 2019. Fast Transformer Decoding: One Write-Head is All You Need. *arXiv preprint arXiv:1911.02150* (2019).

[11] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient Streaming Language Models with Attention Sinks. In *ICLR*.