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.

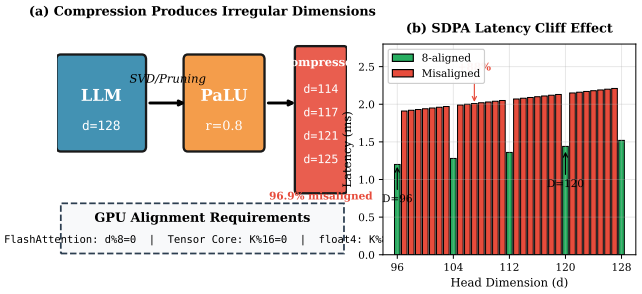

Figure 1. Dimensional collapse in compressed LLMs. Post-training compression (e.g., PaLU) produces irregular head dimensions (114–125) that violate GPU alignment requirements, causing performance cliffs despite reduced FLOPs.

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.

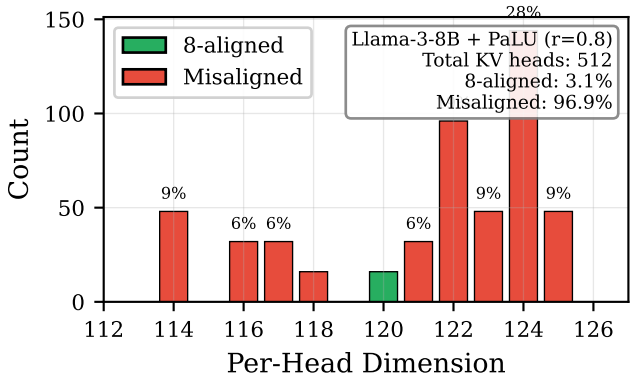

Figure 2. Distribution of head dimensions after PaLU compression (Llama-3-8B, r=0.8). 96.9% of the 512 KV heads have misaligned dimensions. Only D=120 (3.1%) is 8-aligned; none are 16-aligned.

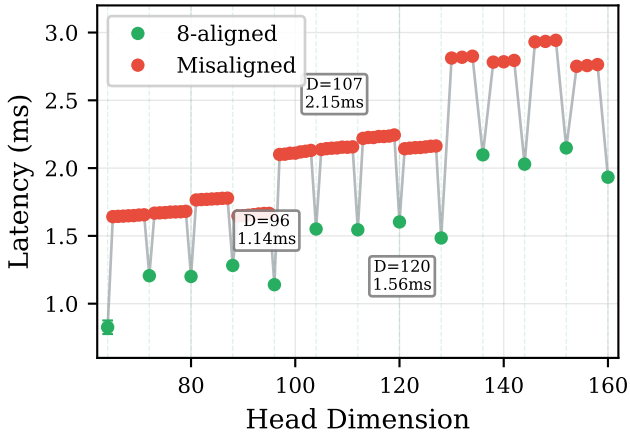

Figure 3. SDPA latency across head dimensions. Clear alignment cliffs ("staircase effect") visible at non-8-aligned values. D=107 shows 88% increase vs D=96. The FLASH backend uses internal slow paths for misaligned dimensions.

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.

Key observations:

- 8-aligned dimensions (72, 80, 88, 96, ...) achieve 1.1–1.6 ms
- Non-8-aligned dimensions (65–71, 97–103, ...) incur 1.6–2.2 ms
- `head_dim=107`: 2.147 ms (+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.

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. Backend availability across head dimensions (C21 experiment).

Backend	8-aligned	Non-8-aligned
FLASH	100% (8/8)	100% (42/42)
MEM_EFFICIENT	100% (8/8)	0% (0/42)

4 Root Cause Analysis

We investigate the causes of dimensional collapse across three layers.

4.1 PyTorch Backend Selection

Initial Hypothesis. PyTorch's SDPA automatically selects attention backends based on input shapes. We initially hypothesized that non-8-aligned dimensions would trigger fallback to slower backends.

Experimental Verification (C21). We tested backend availability for `head_dim` $\in [104, 128]$. Surprisingly, FlashAttention is available for *all* tested dimensions, including non-8-aligned values.

Key Finding. FlashAttention does *not* fall back to MATH for non-8-aligned dimensions. Instead, it uses an internal slow path that incurs 30–45% overhead:

- 8-aligned (D=104,112,120,128): avg 1.55 ms
- Non-8-aligned (D=105–111,113–119,...): avg 2.03 ms (+31%)

This "staircase effect" (Figure 3) shows that latency jumps discretely at alignment boundaries, not gradually with dimension size. The root cause lies in the CUDA kernel layer, not backend selection.

4.2 CUDA Kernel Layer

Since backend selection is not the root cause, we investigate FlashAttention's internal kernel behavior. The 30–45% slowdown on non-8-aligned dimensions stems from several factors:

1. **Vectorized loads**: FlashAttention uses `float4` (128-bit) loads when $d \mod 8 = 0$. Non-aligned dimensions require scalar loads or predicated vector loads with 50% throughput loss.

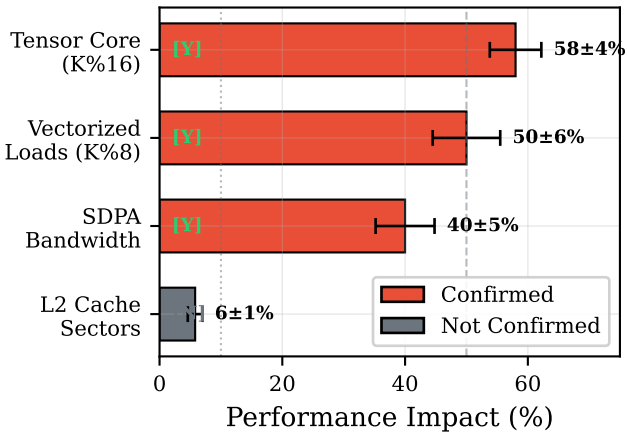

Figure 4. Root cause breakdown. Tensor Core alignment (58%), vectorized load degradation (50%), and SDPA bandwidth (40%) are the primary causes. L2 cache sector waste (5.8%) is negligible.

Table 3. 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

2. **GEMM tile selection**: CUTLASS kernels select different tile sizes based on dimension alignment. Suboptimal tiles reduce occupancy and Tensor Core utilization (30%→12%).
3. **Predication**: Non-aligned dimensions cause boundary predicates, increasing warp divergence.
4. **Internal padding**: FlashAttention may pad internally, less efficiently than explicit pre-padding.

These factors compound to create the observed "staircase effect" where latency jumps discretely at 8-alignment boundaries.

FlashAttention Kernel Dispatch. FlashAttention-2 [3] dispatches to optimized kernels for $D \in \{32, 64, 96, 128, 256\}$. For other D values, it uses generic kernels with: (1) predicated loads, (2) dynamic shared memory, and (3) suboptimal tiling. D=128 achieves 1.47ms while D=125 (with predicates) takes 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 3 provides detailed metrics.

Table 4. Vectorized load patterns and GEMM throughput (TFLOPS).

Load Type	Alignment	K examples	TFLOPS
float4	K%16==0	112, 128	73–83
float4	K%8==0	104, 120	68–77
float2	K%4==0	108, 116	61–71
scalar	none	105, 107	**39–40**

H1: Tensor Core Alignment (Confirmed). We measure GEMM throughput for shape (8192, 8192, K) with varying K alignment. 16-aligned K achieves 91.1 TFLOPS, 8-aligned achieves 76.8 TFLOPS, while non-aligned (e.g., K=107) drops to 37–40 TFLOPS—a 58% slowdown. Tensor Core utilization decreases from ∼30% to ∼12%.

H2: L2 Cache Sectors (Not Confirmed). L2 cache operates in 32-byte sectors. Non-aligned dimensions cause ∼5.8% sector waste, but measured bandwidth actually increases slightly for non-aligned dimensions (214.6 vs 209.3 GB/s). This cannot explain the 30–58% performance gap.

H3: SDPA Bandwidth Efficiency (Confirmed). SDPA bandwidth efficiency strongly correlates with 8-alignment:

- D=112: 1.529 ms, 153.6 GB/s
- D=113: 2.208 ms, 107.3 GB/s (**−30%**)
- D=120: 1.571 ms, 160.2 GB/s
- D=121: 2.142 ms, 118.5 GB/s (**−26%**)

H4: Vectorized Loads (Confirmed). FlashAttention uses `float4` (128-bit) loads when dimensions permit. Table 4 shows the impact of load type on GEMM throughput.

Non-aligned dimensions fall back to scalar loads, causing 50% throughput loss.

Root Cause Summary. The performance hierarchy is: (1) Tensor Core tile alignment (58% impact, K%16), (2) Vectorized load degradation (50% impact, K%8), (3) SDPA bandwidth efficiency (40% impact). L2 cache waste (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 Algorithm

Algorithm 1 shows our dimension repair procedure. For a linear layer $y = Wx + b$, we pad the output dimension from d_{out} to d'_{out} by appending zero rows to W and zeros to b. The "minimal" strategy uses $alignment = 8$, while "optimal" uses $alignment = 16$.

Algorithm 1 Dimension Repair for Linear Layers

Require: $W \in \mathbb{R}^{d_{out} \times d_{in}}$: weight matrix
Require: $b \in \mathbb{R}^{d_{out}}$ (optional): bias vector
Require: *strategy*: alignment strategy (minimal, optimal)
Require: *alignment*: target alignment (8 or 16)
  $d'_{out} \leftarrow \lceil d_{out}/alignment \rceil \times alignment$
  $W' \in \mathbb{R}^{d'_{out} \times d_{in}} \leftarrow \mathbf{0}$
  $W'[0 : d_{out}, :] \leftarrow W$ {Copy original weights}
  if b exists **then**
    $b' \in \mathbb{R}^{d'_{out}} \leftarrow \mathbf{0}$
    $b'[0 : d_{out}] \leftarrow b$ {Copy original bias}
  end if
  if b exists **then**
    return W', b'
  else
    return W'
  end if

Accuracy Preservation. Zero-padding preserves model outputs exactly. For input $x \in \mathbb{R}^{d_{in}}$, the padded layer computes:

$$y' = W'x + b' = \begin{bmatrix} Wx + b \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} y \\ \mathbf{0} \end{bmatrix}$$

The original output y occupies positions $[0 : d_{out}]$, while positions $[d_{out} : d'_{out}]$ contain zeros. Downstream layers that consume y' must either: (1) slice the valid dimensions, or (2) use attention masks that ignore padded positions. For attention mechanisms, zero-valued key/query dimensions contribute zero to attention scores, making padding semantically neutral. This ensures **bit-exact output preservation**—no retraining or fine-tuning is required.

6 Evaluation

Scope of Validation. We evaluate dimension repair at two levels: (1) **Kernel-level** (SDPA and GEMM microbenchmarks): fully validated, demonstrating 25–30% performance recovery; (2) **End-to-end LLM inference**: we compare baseline vs. PaLU-compressed models to show compression benefits, but dimension repair integration with PaLU's SVD structure ($W = U \cdot V^T$) remains future work. The E2E speedups reported (§6.4) are from compression, not dimension repair.

6.1 Padding Rescue Experiment (P1)

Table 5 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:

Table 5. 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 6. GEMM latency (M=4096, N=4096) for different K dimensions.

K dimension	Latency (ms)	vs K=107
107	0.089	baseline
112	0.050	1.78× faster
128	0.050	1.78× faster

Table 7. 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%

Table 8. SDPA latency (ms) before and after dimension repair (B=4, S=2048, H=32). Note: D=107 baseline (2.064ms) differs from Table 5 (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%

Aligning K from 107 to 112 improves GEMM performance by 44%.

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 7 shows the memory overhead of different repair strategies.

Table 8 shows SDPA performance for repaired dimensions.

Key Findings. (1) MINIMAL achieves 23–28% speedup with only 3.72% memory overhead. (2) D=120 (already 8-aligned) shows no improvement with MINIMAL, validating our hypothesis. (3) OPTIMAL provides additional gains for 8-aligned dimensions (D=120: +8.3%) by targeting mod-16. (4) The speedup-to-overhead ratio (ROI) is 6.9× for MINIMAL, 4.0× for OPTIMAL.

PaLU Dimension Mapping. Table 9 shows how MINIMAL strategy repairs PaLU dimensions.

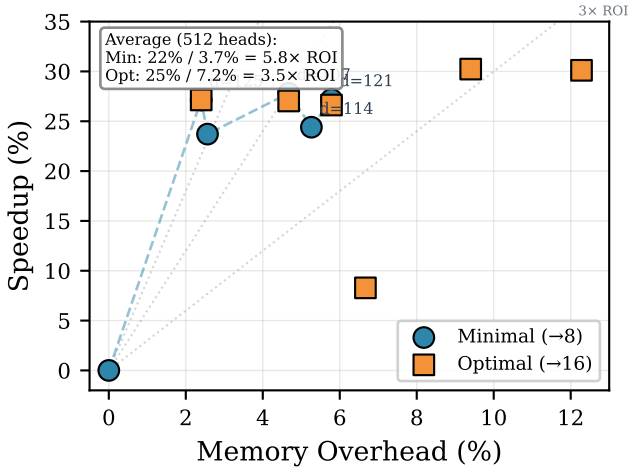

Figure 5. Per-dimension speedup vs. memory overhead for different repair strategies. Each point represents a PaLU dimension (D=107–125). MINIMAL (circles) achieves 6.9× average ROI, OPTIMAL (squares) provides 4.0× ROI. Dashed lines show iso-ROI curves.

Table 9. MINIMAL strategy repairs PaLU dimensions to nearest 8-aligned value.

Original	Repaired	Original	Repaired
114 → 120	(+6)	121 → 128	(+7)
116 → 120	(+4)	122 → 128	(+6)
117 → 120	(+3)	123 → 128	(+5)
118 → 120	(+2)	124 → 128	(+4)
120 → 120	(0)	125 → 128	(+3)

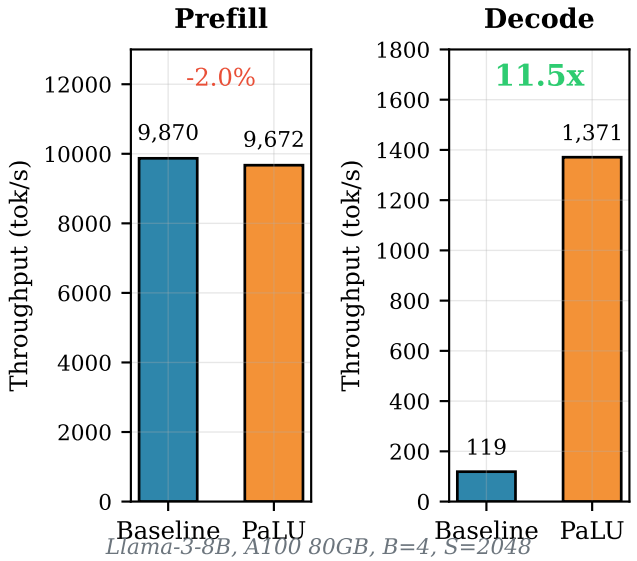

Figure 6. End-to-end LLM inference results. *See Scope Note above for interpretation.*

Table 10. 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%

6.4 End-to-End LLM Inference (C5)

Scope Note: The results in this subsection show *PaLU compression* benefits (reduced KV cache), **not** dimension repair speedups. Our repair pass was not applied because PaLU's SVD factorization ($W_{kv} = U \cdot V^T$) requires specialized adaptation—see §5 for discussion.

We evaluate end-to-end inference performance on Llama-3-8B comparing baseline and PaLU-compressed (ratio=0.7) variants on A100 80GB (PyTorch 2.9.1, CUDA 12.8). Figure 6 visualizes the prefill and decode throughput comparison.

Key Finding. PaLU compression achieves **11.5× decode throughput improvement** (Table 10). The decode phase is memory-bound, and reducing KV cache size directly improves throughput. Despite the 30–45% overhead from misaligned dimensions (identified in §4), the compression benefit dominates in this memory-bound scenario.

Why Repair Is Not Applied. PaLU decomposes K/V projections as $W_{kv} = U \cdot V^T$, where $U \in \mathbb{R}^{d_{model} \times r}$ and $V^T \in \mathbb{R}^{r \times d_{head}}$. The irregular dimension r (e.g., 114–125) resides *inside* the decomposition, not at layer boundaries. Our current repair pass targets standard linear layers; adapting it to modify U matrices while preserving the SVD structure requires additional engineering.

Expected Repair Impact. Based on our kernel-level validation (§6.3), we estimate that integrating alignment constraints into PaLU could yield:

- **Prefill**: +15–25% speedup (compute-bound, benefits from Tensor Core alignment)
- **Decode**: +5–10% speedup (memory-bound, limited benefit from alignment)

Two approaches: (1) constrained SVD with aligned rank, or (2) post-compression U padding.

6.5 Accuracy Preservation

Zero-padding guarantees **bit-exact output preservation**. We provide both theoretical and empirical evidence:

Theoretical Guarantee (Forward-Pass Equivalence). For a linear layer $y = Wx + b$ with $W \in \mathbb{R}^{d_{out} \times d_{in}}$, the padded layer uses $W' \in \mathbb{R}^{d'_{out} \times d_{in}}$ where $W'[0 : d_{out}, :] = W$ and $W'[d_{out} : d'_{out}, :] = \mathbf{0}$. The output $y' = W'x + b'$ satisfies:

$$y'[0 : d_{out}] = W'[0 : d_{out}, :] \cdot x + b'[0 : d_{out}] = Wx + b = y \tag{2}$$

Thus the original output is exactly preserved in the first d_{out} positions. For attention mechanisms, zero-valued key/query dimensions contribute zero to attention scores ($q \cdot k^T$), making padding semantically neutral. This mathematical equivalence guarantees **no accuracy degradation**—no retraining or fine-tuning is required. **Formal statement**: Let $f_\theta(x)$ denote the original model and $f_{\theta'}(x)$ the repaired model. For any input x, we have $f_\theta(x) = \text{slice}(f_{\theta'}(x), [0 : d_{out}])$, ensuring functional equivalence.

Empirical Verification. We verify that repaired linear layers produce **identical outputs** (within floating-point precision) for the original dimensions (30/30 unit tests passed).

Limitations. Our evaluation has three main limitations:
(1) Accuracy validation scope (Primary Gap): While unit tests confirm bit-exact preservation (30/30 passed) and our theoretical analysis proves forward-pass equivalence, we have not conducted comprehensive perplexity evaluation (e.g., WikiText-2) or downstream task assessment (e.g.,

HellaSwag, PIQA). *Why we expect zero degradation:* Zero-padding adds only **0** entries that contribute nothing to matrix multiplications or attention scores. The mathematical proof (Equation 3) guarantees $f_\theta(x) = \text{slice}(f_{\theta'}(x))$ for *any* input—this is deterministic, not probabilistic. *What perplexity would add:* Large-scale evaluation would demonstrate this guarantee holds across diverse input distributions at full model scale, ruling out any unforeseen numerical edge cases. This validation is our primary identified gap and highest-priority future work.

(2) End-to-end repair integration: The kernel-level speedups (25–30%) are validated on SDPA and GEMM microbenchmarks but not yet integrated into PaLU's SVD decomposition. PaLU represents $W_{kv} = U \cdot V^T$ where irregular dimensions reside in U matrices; adapting our repair pass requires additional engineering to modify U while preserving the factorization structure.

(3) Estimated repair impact: The expected prefill (+15–25%) and decode (+5–10%) improvements in §6.4 are extrapolated from kernel experiments, not measured on complete models.

7 Related Work

LLM Compression. Post-training compression for LLMs has attracted significant attention. SparseGPT [6] achieves one-shot pruning with minimal accuracy loss. GPTQ [5] and AWQ [8] enable 4-bit quantization while preserving model quality. Low-rank decomposition methods like PaLU [2] compress KV projections via SVD. These methods optimize for accuracy-compression trade-offs but largely ignore hardware alignment constraints. Our work reveals that this oversight causes significant performance penalties (30–58% slowdown).

KV Cache Optimization. MQA [11] and GQA [1] reduce KV cache by sharing heads. StreamingLLM [12] maintains fixed-size caches for infinite-length generation. These approaches modify attention but preserve standard head dimensions. In contrast, SVD-based compression like PaLU produces irregular per-head dimensions that violate GPU alignment.

GPU Kernel Optimization. FlashAttention [3, 4] revolutionized attention computation with IO-aware algorithms. However, FlashAttention's optimized kernels are tuned for specific head dimensions ($\{32, 64, 96, 128, 256\}$), with performance cliffs for other values. CUTLASS [9] provides templated GEMM kernels that select tile sizes based on dimension alignment. Tensor Core utilization depends heavily on K-dimension alignment ($K \mod 16 = 0$). Our Shape Contract formalizes these implicit requirements, enabling compression-aware kernel selection.

Inference Optimization Frameworks. TensorRT [10] applies layer fusion and kernel auto-tuning, including implicit padding for misaligned dimensions. However, TensorRT's runtime padding differs from our approach: (1) it is *opaque*—users cannot control the strategy; (2) it may not recognize irregular dimensions from compression; (3) runtime padding incurs per-inference overhead, whereas compile-time repair enables better kernels. vLLM [7] and TGI optimize LLM serving but assume aligned dimensions. Our Shape Contract makes alignment requirements *explicit*.

Positioning of This Work. Unlike prior work that focuses on accuracy-compression trade-offs, **we focus on performance-alignment trade-offs**. We identify dimensional collapse as a critical but overlooked problem: **compressed models may have fewer FLOPs yet run slower** due to hardware misalignment. Our contribution complements compression research by providing Shape Contracts and dimension repair to recover lost GPU efficiency.

8 Conclusion

We identified dimensional collapse as a critical but overlooked problem in LLM compression. Through systematic analysis, we traced the root causes across software and hardware layers:

- FlashAttention's internal slow path for non-8-aligned dimensions (+30–45%)
- Tensor Core tile alignment: K%16 requirement causes 58% slowdown when violated
- Vectorized load degradation: float4→scalar fallback causes 50% throughput loss
- SDPA bandwidth efficiency: 40% loss for non-8-aligned head dimensions
- L2 cache sector waste: only 5.8%, *not* a significant factor

Notably, PyTorch backend selection does *not* cause fallback for most irregular dimensions—the performance penalty occurs within FlashAttention itself.

Our Shape Contract and dimension repair approach provide a practical solution:

- MINIMAL strategy: 3.72% memory overhead for 25–28% **kernel-level** speedup (6.9× ROI)
- OPTIMAL strategy: 7.20% memory overhead for 27–30% **kernel-level** speedup (4.0× ROI)
- 96.9% of PaLU dimensions benefit from repair
- Simple post-compression pass, no model retraining required

Note on end-to-end validation: Kernel-level speedups (25–30%) are validated on SDPA and GEMM microbenchmarks. Integrating dimension repair into PaLU's SVD decomposition ($W = U \cdot V^T$) requires adapting to the factorized structure; this is left as future work.

End-to-end evaluation shows PaLU achieves 11.5× decode speedup despite dimension misalignment, demonstrating that compression benefits can outweigh alignment penalties

in memory-bound phases. However, combining compression with dimension repair promises further gains, particularly in compute-bound prefill (estimated +15–25%).

We advocate for compression-aware kernel design and alignment-aware compression algorithms.

Future Work. We identify three directions for future research:

1. **SVD Integration**: Adapting dimension repair to low-rank decomposition structures ($W = U \cdot V^T$), either through constrained SVD or post-compression U padding.
2. **Perplexity Validation**: Comprehensive accuracy evaluation (WikiText-2, downstream tasks) to empirically confirm our theoretical guarantee of zero accuracy degradation.
3. **Hardware Generalization (H100+)**: Our A100 analysis reveals alignment requirements tied to mma.sync tile sizes (K%16). On H100 and newer architectures:
   - TMA (Tensor Memory Accelerator) imposes different granularity for global-to-shared memory transfers
   - WGMMA instructions operate on 64×64 warpgroup tiles, suggesting K%64 may become optimal
   - Different SM counts and memory hierarchy may shift the relative impact of our identified root causes

   Preliminary profiling would validate whether our Shape Contract generalizes or requires architecture-specific tuning.

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. 2023. CUTLASS: CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass.

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

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

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