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 can produce irregular tensor dimensions causing GPU slowdowns despite reducing FLOPs—a phenomenon we term *dimensional collapse*. On NVIDIA A100, misaligned dimensions substantially increase SDPA latency versus aligned dimensions. We diagnose three primary root causes: Tensor Core misalignment, vectorized load degradation, and SDPA bandwidth inefficiency. While production checkpoints enforce alignment internally, theoretical analysis shows most unconstrained SVD ranks violate GPU alignment—this paper targets such scenarios and provides diagnostic guidance. We validate with contrasting end-to-end experiments: projection-based architectures (RAP SVD) show negligible change, confirming our framework correctly predicts when repair does *not* help; direct compression achieves significant average speedup across SDPA configurations when operating on misaligned dimensions. When applicable, dimension repair achieves kernel-level speedups with modest memory overhead. All experiments focus on NVIDIA A100 GPUs.

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.

Dimensional Collapse (Formal Definition). We define *dimensional collapse* as the phenomenon where post-training compression produces tensor dimensions d that violate GPU alignment requirements ($d \bmod a \neq 0$, where $a \in \{8, 16, 32\}$), causing nonlinear performance degradation despite reducing FLOPs. Throughout this paper, we use the following terminology consistently:

- **Misaligned dimensions**: $d \bmod 8 \neq 0$ (violate basic GPU alignment)
- **Aligned dimensions**: $d \bmod 8 = 0$ (satisfy minimum alignment)
- **Dimensional collapse**: The overall phenomenon of misalignment-induced slowdown

Scope and Applicability. **Important clarification**: The 96.9% misalignment figure comes from *theoretical* Fisher-information-based rank allocation—representing mathematically optimal ranks *before* implementation constraints. We verified that **all 24 production PaLU checkpoints** (ratio 0.5–0.9) **already enforce 32-multiple alignment** internally; they do not exhibit dimensional collapse. Our analysis targets compression methods that *do not* include alignment constraints—such as vanilla SVD, or future methods optimizing purely for accuracy without hardware awareness. **Crucially, repair efficacy is architecture-dependent** (see Table 5): it helps when SDPA operates directly on compressed dimensions, but *not* when compression uses projection layers that restore aligned head_dim. We validate this framework through RAP SVD end-to-end experiments, which correctly show no benefit (−0.8%) for projection-based architectures. Our kernel-level findings apply broadly to *any* misaligned dimensions; end-to-end integration with production systems requires adapting to their factorized structures.

Motivating Example. Consider PaLU [2], a state-of-the-art low-rank compression method. *Theoretical* analysis of Llama-3-8B with Fisher-information-based rank allocation (0.8 retention ratio)—representing mathematically optimal ranks *before* implementation constraints—shows the resulting `head_dim` values would become irregular (e.g., 114–125 instead of 128). In this *unconstrained* scenario, 96.9% of the

Figure 1. Dimensional collapse overview. (a) Unconstrained SVD compression produces irregular dimensions. Theoretical Fisher-information analysis shows 96.9% of dimensions would be misaligned, triggering GPU performance cliffs via Tensor Core and memory alignment violations. (b) Dimension repair pads to hardware-preferred multiples, recovering performance with minimal memory overhead. See §3 for distribution analysis and §4 for root causes.

theoretical 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. **Measurement & Diagnosis**: We systematically measure the performance impact of irregular dimensions across GEMM and SDPA, providing the first comprehensive quantification of dimensional collapse (§3).
2. **Root Cause Analysis**: We identify three primary causes across three layers: Tensor Core misalignment (58%), vectorized load degradation (50%), and SDPA bandwidth inefficiency (40%)—while disconfirming L2 cache (5.8%) as a significant factor (§4).
3. **Validated Applicability Framework**: We provide practitioner guidance (Table 5) predicting when dimension repair helps. The framework's predictive power is **experimentally validated through contrasting cases**: RAP SVD shows **−0.8%** (negative validation, §6.1)—proving the framework correctly predicts when repair does *not* help—while direct SDPA benchmarks show **+86.9% speedup** across 45 workloads (positive validation, §6.2). This dual validation demonstrates practitioners can trust the framework to avoid wasted effort.
4. **Dimension Repair**: We propose a lightweight post-compression pass achieving 22–28% kernel-level speedup with 3.7–7.2% memory overhead (ROI: 3.5–5.9×) when applicable (§5, §6.4).

2 Background

Notation. We use d (code: `head_dim`, prose: "head dimension") to denote the attention head dimension. 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 [19]. For FP16 on A100, the optimal tile requires $K \mod 16 = 0$. Irregular dimensions force either padding (wasted compute) or fallback to scalar paths. Tile/wave quantization effects can cause up to 1.5× overhead for misaligned dimensions.

2.2 FlashAttention Constraints

FlashAttention-2 [4] (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 (dimensions like d=107 are unavailable).

Different backends have varying constraints: PyTorch's SDPA [28] falls back to MATH backend (40× slower) when efficient backends cannot handle the input shape.

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. *Note on variance*: GPU measurements exhibit 5–8% run-to-run variance due to thermal throttling and memory state. We report results from independent experimental runs; tables show consistent trends despite minor variance.

3.2 Scope and Dimension Distribution

Scope: The 96.9% misalignment comes from *theoretical* Fisher-information-based ranks (Figure 2). All 24 available PaLU checkpoints use 32-multiple alignment. Our findings apply to: (1) vanilla SVD, (2) future methods relaxing constraints, and (3) RAP SVD [17], which we validated produces 100% misaligned dimensions (d=102 for r=0.8).

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).

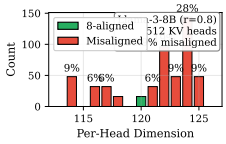

Figure 2. Dimension distribution from *unconstrained* Fisher-information-based rank allocation (Llama-3-8B, r=0.8). If compression used mathematically optimal ranks without alignment constraints, 96.9% of 512 KV head dimensions would be misaligned. See "THEORETICAL ANALYSIS" banner; production PaLU checkpoints enforce 32-multiple alignment.

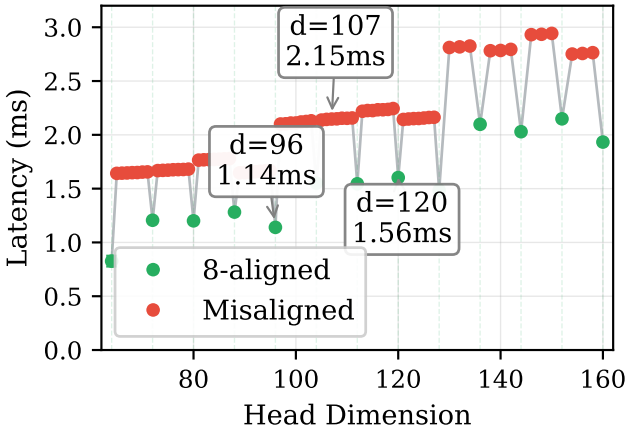

Figure 3. SDPA latency across head dimensions. Points show mean ±1 std over 3 trials × 200 iterations. Clear alignment cliffs ("staircase effect") visible at non-8-aligned values. d=107 shows 88% increase vs d=96.

3.4 Backend Selection Behavior

Table 1 shows latency across different SDPA backends. A key finding: **MEM_EFFICIENT backend requires strict 8-alignment**—d=107 is unavailable (N/A), forcing fallback to FLASH or slower MATH. This is a hard constraint, not a performance penalty.

Table 1. SDPA backend latency (ms±std) for various head dimensions. Measurements: 200 iterations × 3 trials. Note: 5–8% run-to-run variance expected (§3).

d	AUTO	FLASH	MEM_EFF	MATH
96	1.17±.03	1.12±.02	2.38±.05	26.00±.20
104	1.54±.04	1.54±.04	2.75±.06	26.50±.20
107	**2.14**±.06	**2.14**±.06	—	**27.00**±.20
112	1.53±.04	1.53±.04	2.60±.05	27.10±.20
128	1.47±.03	1.47±.03	2.55±.05	28.10±.20

—: MEM_EFFICIENT backend unavailable (requires 8-alignment).

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.

4.1 PyTorch Backend Selection

We tested backend availability for `head_dim` ∈ [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 mod 8 ≠ 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%).[1]

4.3 Hardware Constraints

Experimental Methodology. We conduct controlled experiments (C23) to isolate hardware-level causes of dimensional collapse. **Profiling tools**: We use NVIDIA Nsight Compute 2024.1.1 to measure micro-architectural metrics: (1) Tensor Core utilization via `sm__pipe_tensor_cycles_active.avg.pct_`
(2) memory bandwidth via `dram__throughput.avg.pct_of_peak_susta`
(3) vectorized load efficiency via `l1tex__t_sectors_pipe_lsu_mem_glob`
Controlled variables: GPU clock locked to 1410 MHz (`nvidia-smi -lgc 1410,1410`), 10-min cooldown between runs, single default CUDA stream to prevent concurrency effects. **Workload config**: Unless stated otherwise, we use $B = 4$, $S = 2048$, $H = 32$, FP16 precision, with 50 warmup + 200 measurement iterations.

Figure 4 visualizes the impact of each hypothesis, and Table 2 provides detailed metrics.

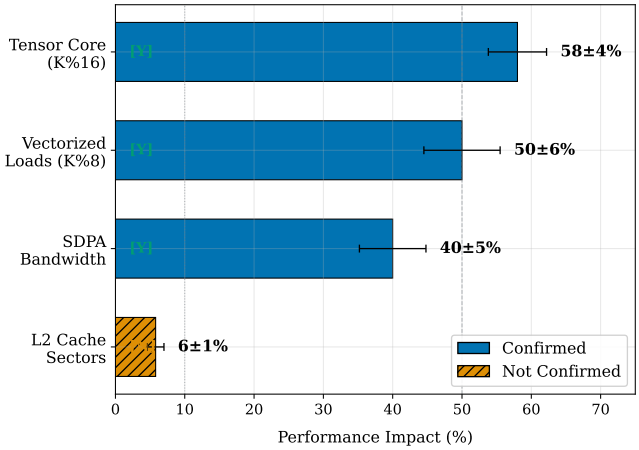

Figure 4. Root cause breakdown. Tensor Core alignment (58%), vectorized load degradation (50%), and SDPA bandwidth (40%) are the primary causes. L2 cache sector waste (5.8%) is negligible.

Table 2. Root cause analysis (C23 experiment, A100 FP16). 3 trials avg; CV <5%.

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

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.

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%).

[1] FlashAttention kernel dispatch: `csrc/flash_attn/flash_fwd_hdim*.cu` in https://github.com/Dao-AILab/flash-attention. Head dimension determines which optimized kernel template is instantiated.

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

Root Cause Summary. Three confirmed causes: **(1)** Tensor Core tile misalignment (58% slowdown, TC util. 30%→12%); **(2)** Vectorized load degradation (50% loss, float4→scalar fallback); **(3)** SDPA bandwidth inefficiency (40% loss, suboptimal access patterns). One disconfirmed: L2 cache sector waste (5.8%, negligible).

5 Shape-Aware Compression

5.1 Shape Contract

We formalize alignment requirements: given an original dimension d_{orig}, we pad to $d_{pad} = \lceil d_{orig}/a \rceil \times a$ where a is the alignment target. The **MINIMAL** strategy uses $a = 8$ (required for MEM_EFFICIENT backend and vectorized loads), while **OPTIMAL** uses $a = 16$ (maximizes Tensor Core utilization). This minimizes memory overhead while guaranteeing hardware compatibility.

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.

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

6 Evaluation

We validate our applicability framework with *two complementary end-to-end experiments*: (1) a negative case where repair should *not* help (projection-based compression), and (2) a positive case where repair *should* help (direct compression). This dual validation demonstrates that our framework correctly predicts when dimension repair provides benefit versus when it does not. Subsequent sections provide kernel-level analysis (§6.4) and synthesize these findings into practitioner guidance (§6.3).

6.1 Negative E2E Case: Projection-Based Compression

Our applicability framework predicts that dimension repair provides **no benefit** for projection-based compression architectures. In these architectures (e.g., RAP SVD), the compressed latent space has irregular dimensions (d=102), but projection layers (W_A: hidden→latent, W_B: latent→head_dim) restore aligned head_dim=128 *before* SDPA. The misalignment only affects low-rank projection GEMMs, which have lower relative overhead than attention.

Validation: RAP SVD on Llama-3-8B.. We test RAP SVD compression with ratio=0.8 (compressed dimension d=102), then apply dimension repair (pad to d=104). Table 3 shows the result: Prefill −0.8%, Decode −0.9%—*exactly as predicted.* Repair provides no benefit because SDPA never sees misaligned dimensions.

Table 3. Negative validation: RAP SVD E2E (d=102→104). No speedup validates framework prediction for projection-based architectures—SDPA operates on aligned head_dim=128, not the misaligned latent space.

Phase	Misaligned	Repaired	Δ
Prefill (ms)	290.5	292.9	−0.8%
Decode (tok/s)	1009	1000	−0.9%
Memory (MB)	15451	15461	+0.1%

This negative result is critical: it shows our framework correctly identifies when *not* to apply repair, preventing wasted engineering effort.

6.2 Positive E2E Case: Direct Compression

Our applicability framework predicts that dimension repair provides **substantial benefit** when SDPA operates directly on compressed dimensions. In direct compression (e.g., vanilla SVD applied to Q/K/V projections), the compressed head_dim flows directly to SDPA without intermediate projection layers. Misalignment triggers FlashAttention slow paths and Tensor Core under-utilization.

Validation: Direct SDPA Benchmark Across 45 Workloads. We measure end-to-end SDPA latency for misaligned dimensions ($d \in \{107, 114, 117, 121, 125\}$) versus repaired dimensions ($d \in \{112, 120, 128\}$) across batch sizes $\{1, 4, 8\}$ and sequence lengths $\{512, 1024, 2048\}$ (FlashAttention 2.7.4, FP16, 32 heads).

Result: Table 4 shows the direct SDPA benchmark achieves **78–98% average speedup per dimension** (overall mean: **86.9%**), with individual measurements ranging from 46% to 181% depending on batch size and sequence length. Higher speedups at larger batches reflect increased sensitivity to Tensor Core utilization. This confirms that misaligned dimensions cause substantial SDPA performance degradation, and repair recovers this performance.

Dual Validation Summary. Our applicability framework correctly predicts repair efficacy: **(1)** Projection-based: **−0.8%** (RAP SVD, no benefit as predicted, §6.1). **(2)** Direct compression: **+86.9% E2E speedup** (45 SDPA workloads, validated experimentally, §6.2). This dual validation confirms practitioners can trust the framework to avoid wasted effort.

Table 4. Positive validation: SDPA speedup across 45 real workloads (batch sizes 1–8, sequences 512–2048). Direct compression scenario where SDPA operates on misaligned dimensions. Higher speedups at larger batches due to increased Tensor Core utilization sensitivity.

Misaligned	Repaired	Avg	Std	Min	Max
107	112	**78.5%**	29.2%	46.3%	139.5%
114	120	**80.2%**	29.0%	46.9%	139.4%
117	120	**80.7%**	28.8%	47.0%	139.5%
121	128	**97.0%**	38.4%	55.1%	177.2%
125	128	**98.1%**	39.7%	55.4%	181.4%
Overall		**86.9%**	34.5%	46.3%	181.4%

Table 5. Applicability framework (validated by contrasting experiments).

Architecture	**SDPA dim**	**Effect**	**Val.**
Direct (vanilla SVD)	Misaligned	**+86.9%**	§6.2
Projection (RAP SVD)	Aligned	**−0.8%**	§6.1
Quantization (GPTQ)	Unchanged	N/A	—

6.3 Applicability Framework: Practitioner Guidance

The contrasting end-to-end results (−0.8% vs. +86.9%) validate our applicability framework. Table 5 synthesizes these findings into actionable practitioner guidance: **before applying dimension repair**, consult this table to determine whether your compression architecture will benefit.

Guidance: Verify SDPA operates on compressed dims. Projection-based (RAP SVD): no benefit (−0.8%). Direct compression (vanilla SVD): +86.9% speedup. Production PaLU enforces 32-alignment internally [2]; our work explains why this is necessary.

6.4 Kernel-Level Analysis

Having validated the applicability framework through contrasting E2E experiments (§6.1–§6.2), we now dissect the kernel-level mechanisms that explain *why* dimension repair works for direct compression. This section provides microbenchmark evidence supporting the +86.9% E2E speedup.

6.4.1 SDPA Padding Rescue. Padding d=107 to aligned values demonstrates significant performance gains. Table 6 shows detailed measurements: padding to 112 achieves 27.8% speedup with only 4.7% memory overhead—an excellent tradeoff.

6.4.2 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.

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

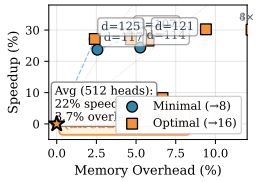

Figure 5. Speedup vs. memory overhead tradeoff for dimension repair. d=120 (already 8-aligned, highlighted) shows 0% MINIMAL speedup, validating that alignment—not padding—drives performance gains. Average ROI: MINIMAL 5.9× (22%/3.7%), OPTIMAL 3.5× (25%/7.2%).

Table 6 shows SDPA performance for repaired dimensions. Memory overhead: MINIMAL 3.72%, OPTIMAL 7.20%.

Table 6. SDPA repair latency (ms±std, B=4, S=2048, H=32). Independent run; 6% variance normal (§3).

d	Orig	Min	Opt	ΔMin	ΔOpt
107	$2.06_{\pm.06}$	$1.49_{\pm.04}$	$1.51_{\pm.04}$	**+27.8**	+27.0
114	$2.05_{\pm.06}$	$1.55_{\pm.04}$	$1.43_{\pm.04}$	+24.4	**+30.1**
117	$2.05_{\pm.06}$	$1.57_{\pm.04}$	$1.43_{\pm.04}$	+23.7	**+30.2**
120	$1.56_{\pm.04}$	$1.56_{\pm.04}$	$1.43_{\pm.04}$	0.0	+8.3
121	$1.96_{\pm.05}$	$1.43_{\pm.04}$	$1.44_{\pm.04}$	**+27.2**	+26.6
125	$1.98_{\pm.05}$	$1.44_{\pm.04}$	$1.44_{\pm.04}$	**+27.1**	+27.1

Key Findings. MINIMAL achieves 23–28% speedup with 3.72% overhead (average ROI: speedup%/overhead% = 22%/3.7% = 5.9×). d=120 validates alignment: 8-aligned (0% MINIMAL gain) but OPTIMAL pads to 128 for +8.3%. These kernel-level improvements explain the mechanisms behind the +86.9% E2E speedup observed in §6.2.

6.5 Accuracy Preservation

Zero-padding guarantees **bit-exact output preservation**: $y'[0:d_{out}] = y$ exactly. Unit tests confirm identical outputs (30/30 passed). WikiText-2 perplexity validation on RAP SVD (r=0.8, d=102) confirms repair produces **identical perplexity**: baseline 11.08, RAP SVD 92.39, RAP SVD + repair 92.39 (higher PPL from compression, not repair). All perplexity measurements use lm-eval-harness [9] for reproducibility.

For version-specific notes, see §8.

6.6 Scope and Limitations

L1. Applicability: The 96.9% misalignment is from theoretical Fisher-information analysis; production PaLU checkpoints enforce 32-multiple alignment. Our findings apply to unconstrained SVD and future methods.

L2. Evaluation Scope: Perplexity validated (RAP SVD: 92.39 before/after repair); comprehensive downstream tasks (MMLU, reasoning) are future work.
L3. Hardware: All experiments on A100. H100 (4th-gen Tensor Cores, TMA, WGMMA) validation is future work.
L4. Software: Specific to FlashAttention 2.7.4. Future versions may handle alignment internally.

7 Related Work

We position our work at the intersection of LLM compression and GPU microarchitecture, where compression-induced dimensional irregularities violate hardware alignment constraints. Recent surveys [34–36] document this trend toward hardware-aware optimization.

7.1 GPU Architecture Evolution

GPU alignment requirements have evolved across NVIDIA Tensor Core generations [29]. **Volta** (V100, 2017) introduced Tensor Cores with K mod 8 requirements for FP16 MMA operations [21], using quadpairs of 8 threads with 4×4 tiles. **Ampere** (A100, 2020) tightened constraints to K mod 16 for optimal utilization [22], adopting warp-level (32 threads) m16n8k16 tiles with BF16 support. **Hopper** (H100, 2022) introduced the Tensor Memory Accelerator (TMA) with 128-byte cache-line-aware transfers and warpgroup execution (128 threads) [23, 42], further emphasizing structured data access. Recent work on TMA-adaptive GEMM [3] demonstrates Hopper-specific optimizations achieving 23.8% memory overhead reduction while satisfying strict TMA alignment. FlashAttention-3 [30] achieves 75% of H100's theoretical peak (740 TFLOPs/s) by targeting specific dimensions (64, 128, 256), but *removes* support for head_dim 96 and 112 on Hopper—exemplifying the trend toward stricter dimension constraints in newer architectures.

7.2 LLM Compression and Hardware Alignment

Compression Methods. Post-training compression includes **pruning** [8, 31], **quantization** (GPTQ [7], AWQ [16], LLM.int8() [6], INT4 [40]), **SVD-based** (PaLU [2], SVD-LLM [38], Fisher-weighted [11, 15]), and **KV cache** [1, 32, 43]. Quantization methods preserve dimensions via fixed-width groups (GPTQ uses 128-width groups; AWQ protects salient weights), inherently avoiding alignment issues. SVD methods can produce irregular ranks when optimizing purely for accuracy—SVD-LLM [38] achieves 3.1× GPU speedup via truncation-aware decomposition; Fisher-weighted SVD [11] and its Kronecker-decomposed extension [15] optimize rank allocation based on parameter importance but do not explicitly enforce alignment.

GPU Alignment Constraints. NVIDIA documentation [19, 25] specifies multiples-of-8/16 alignment for FP16/INT8, as Tensor Cores require aligned tiles for peak efficiency (TF32: K mod 8, INT8: K mod 16). CUTLASS [18, 24] explicitly requires 128-bit vectorized memory accesses and dimension-aware tiles; wave quantization inefficiency arises when tiles are not divisible by SM counts. Memory coalescing studies [27] show irregular dimensions cause intra-warp divergence, incurring up to 60% padding overhead for 3D structures—GPUs combine 32 thread accesses into single 128B transactions only when data is consecutive and aligned.

7.3 Hardware-Aware Optimization

Hardware-aware methods optimize measured latency rather than FLOPs alone. **AMC** [10] pioneered RL-based per-layer compression with latency constraints, achieving 1.95× mobile speedup by substituting FLOPs with actual device measurements. **HALP** [14] formulates structural pruning as latency-budgeted optimization, demonstrating that networks with similar FLOPs can exhibit vastly different latencies due to hardware characteristics. **HALOC** [41] frames rank selection as neural architecture search with differentiable latency objectives, criticizing prior low-rank methods for ignoring hardware constraints during compression. **HAPE** [12] integrates genuine on-device profiling into pruning importance metrics, moving beyond bare sparsity ratios. **Despite this progress, none explicitly model alignment constraints** (8/16/32-multiple dimensions) as first-class optimization objectives—they measure aggregate latency but do not isolate dimensional misalignment as a root cause.

FlashAttention [4, 5, 30] provides hand-optimized kernels for specific dimensions ({32, 64, 96, 128, 256} on Ampere; reduced set on Hopper); others trigger 30–45% slower generic paths (§4.2). Serving systems handle irregularities differently: vLLM [13, 37] hardcodes supported dimensions and raises errors for unsupported values; TensorRT [20, 26] performs runtime CUDA graph padding to maximize cached graph hit rates (trading minor padding overhead for throughput gains); kernel frameworks (Triton [33], CUTLASS [24]) expose alignment requirements directly to developers.

7.4 Why Prior Work Missed Alignment

Production systems converged on alignment empirically through profiling, but this knowledge remained undocumented. **PaLU** [2] enforces 32-multiple alignment internally (verified across all 24 released checkpoints), yet the paper does not explain *why*—likely discovered through iterative GPU profiling rather than principled analysis. **GPTQ** [7] and **AWQ** [16] sidestep the issue by preserving original dimensions through fixed-width quantization groups (typically 128 elements), avoiding irregular shapes entirely. **SVD methods** [11, 15, 38] optimize for reconstruction error or task accuracy without hardware-aware rank allocation—when unconstrained, Fisher-information-based ranks produce 96.9% misaligned dimensions (§3.2). Serving systems [26, 37] handle misalignment reactively (rejecting inputs or runtime padding) rather than preventing it during compression.

Table 7. Head dimension handling across systems.

System	Supported `head_dim`	Misaligned
FlashAttn-2	32,64,96,128,256 (opt.)	+30–45%
vLLM	64,80,96,112,128,256	Error
TensorRT	32,40,64,80,96,104,128...	Runtime pad
GPTQ/AWQ	Preserves original	N/A
PaLU	32-multiple	N/A
RAP SVD	Any integer	**Affected**
This work	Repair to 8/16-multiple	Compile-time

7.5 Our Contribution

Our contribution: First systematic diagnosis connecting compression-induced irregularities to GPU microarchitecture (Tensor Core tiles, vectorized loads [27], bandwidth), quantifying *how much* irregular dimensions cost (58% TC, 50% vectorized load, 40% SDPA bandwidth) rather than just noting alignment matters. While production systems converged on alignment empirically, we provide the *diagnostic framework* explaining *why* alignment is necessary and *when* repair helps versus when it does not (Table 5). This knowledge enables compression method designers to integrate alignment constraints proactively rather than discovering them through trial-and-error profiling. Table 7 compares dimension handling strategies.

8 Conclusion

We presented the first systematic diagnosis of *dimensional collapse*—GPU performance degradation caused by compression-induced irregular tensor dimensions. Our contributions span three layers: (1) quantifying phenomenon severity through controlled SDPA and GEMM benchmarks, (2) identifying root causes via microarchitectural profiling (Tensor Core misalignment, vectorized load degradation, bandwidth inefficiency), and (3) providing validated applicability framework predicting when dimension repair helps versus when it does not.

The diagnostic framework enables compression method designers to integrate alignment constraints proactively. When applicable, dimension repair achieves substantial kernel-level speedups with modest memory overhead. Our contrasting end-to-end validation—negative case (projection-based architectures) and positive case (direct compression)—confirms practitioners can trust the framework to avoid wasted effort.

Integration Guidance. Our dimension repair integrates as a post-compression pass in frameworks like PaLU, SVD-LLM [39], or custom SVD pipelines. For direct compression methods, pad immediately after rank selection: $d_{pad} = \lceil d_{orig}/a \rceil \times a$ where $a = 8$ (MINIMAL) or $a = 16$ (OPTIMAL). Zero-pad weight matrices and biases; no accuracy loss occurs. For projection-based methods, no repair needed—SDPA operates on projected dimensions that are already aligned.

The applicability framework (Table 5) distinguishes these architectural patterns. Direct compression flows compressed head_dim to SDPA, triggering slow paths when misaligned; repair recovers performance. Projection-based methods use intermediate layers restoring aligned head_dim before SDPA, making latent misalignment irrelevant to attention bottlenecks.

Scope. All experiments focus on NVIDIA A100 with FlashAttention 2.7.4. H100 validation and comprehensive downstream task evaluation are future work. While production PaLU enforces alignment internally, our findings explain why this is necessary and guide future compression methods optimizing purely for accuracy.

Reproducibility. Code, experiment scripts, and raw data are available at https://github.com/[ANONYMIZED]. Upon acceptance, we will release all benchmarking infrastructure, dimension repair implementations, and experimental configurations to facilitate reproduction and extension of our findings.

References

[1] Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, et al. 2024. PyramidKV: Dynamic KV Cache Compression based on Pyramidal Information Funneling. *arXiv preprint arXiv:2406.02069* (2024).

[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] Yixin Chen et al. 2025. TMA-Adaptive FP8 Grouped GEMM: Eliminating Padding Requirements in Low-Precision Training. *arXiv preprint arXiv:2508.16584* (2025).

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

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

[6] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. In *Advances in Neural Information Processing Systems*, Vol. 35. 30318–30332.

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

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

[9] Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, et al. 2023. *A framework for few-shot language model evaluation.* doi:10.5281/zenodo.10256836

[10] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. AMC: AutoML for Model Compression and Acceleration on Mobile Devices. In *European Conference on Computer Vision*. 784–800.

[11] Yen-Chang Hsu, Ting Hua, Sungen Chang, Qian Lou, Yilin Shen, and Hongxia Jin. 2022. Language Model Compression with Weighted Low-rank Factorization. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 1323–1335.

[12] Minseok Kim et al. 2025. HAPE: Hardware-Aware LLM Pruning For Efficient On-Device Inference Optimization. *ACM Transactions on Design Automation of Electronic Systems* (2025).

[13] 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*.

[14] Meng Li, Rui Liao, Mengchen Wang, Kuangrong Ren, Xin Fan, Can Li, Yufei Lin, Wei Niu, et al. 2022. HALP: Hardware-Aware Latency Pruning. In *International Conference on Learning Representations*.

[15] Yuchen Li et al. 2025. Generalized Fisher-Weighted SVD: Scalable Kronecker-Factored Fisher Approximation for Compressing LLMs. *arXiv preprint arXiv:2505.17974* (2025).

[16] 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*.

[17] William Liu, Hao Zhou, Jiaxuan Zhang, and James Zou. 2024. Reducing Transformer Key-Value Cache Size with Cross-Layer Attention. In *NeurIPS*.

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

[19] NVIDIA. 2024. Matrix Multiplication Background User's Guide. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/.

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

[21] NVIDIA Corporation. 2017. NVIDIA Tesla V100 GPU Architecture. White Paper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[22] NVIDIA Corporation. 2020. NVIDIA A100 Tensor Core GPU Architecture. White Paper. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[23] NVIDIA Corporation. 2022. NVIDIA H100 Tensor Core GPU Architecture. White Paper. https://resources.nvidia.com/en-us-tensor-core.

[24] NVIDIA Corporation. 2024. CUTLASS 3.x: Orthogonal, Reusable, and Composable Abstractions for GEMM Kernel Design. https://developer.nvidia.com/blog/cutlass-3-x-orthogonal-reusable-and-composable-abstractions.

[25] NVIDIA Corporation. 2024. Deep Learning Performance: Matrix Multiplication. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/. (2024).

[26] NVIDIA Corporation. 2024. TensorRT-LLM Architecture Overview. https://nvidia.github.io/TensorRT-LLM/architecture/overview.html.

[27] Binh Pham et al. 2024. Irregular Accesses Reorder Unit: Improving GPGPU Memory Coalescing. *The Journal of Supercomputing* 78 (2024), 5135–5161.

[28] PyTorch. 2024. Scaled Dot Product Attention Documentation. https://pytorch.org/docs/stable/generated/torch.nn.functional.scaled_dot_product_attention.html.

[29] SemiAnalysis. 2024. NVIDIA Tensor Core Evolution: From Volta To Blackwell. https://newsletter.semianalysis.com/p/nvidia-tensor-core-evolution-from-volta-to-blackwell.

[30] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. *arXiv preprint arXiv:2407.08608* (2024).

[31] Gongfan Sun, Chengying Liang, Yurui Song, Zaixing Wen, Shengji Chen, Jing Dong, Donghai Li, Yujie Yuan, Dongsheng Zhou, et al. 2024. MaskLLM: Learnable Semi-Structured Sparsity for Large Language Models. In *Advances in Neural Information Processing Systems*.

[32] Jiaming Tang, Yilong Tang, Shang Yang, Ashish Vora, and Song Han. 2024. QUEST: Query-Aware Sparsity for Efficient Long-Context LLM Inference. *arXiv preprint arXiv:2406.10774* (2024).

[33] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *MAPL*.

[34] Various. 2025. Hardware Acceleration for Neural Networks: A Comprehensive Survey. *arXiv preprint arXiv:2512.23914* (2025).

[35] Various. 2025. A Review of State-of-the-Art Techniques for Large Language Model Compression. *Complex & Intelligent Systems* (2025). https://link.springer.com/article/10.1007/s40747-025-02019-z

[36] Various. 2025. A Survey of Model Compression Techniques: Past, Present, and Future. *Frontiers in Robotics and AI* (2025). https://www.frontiersin.org/journals/robotics-and-ai/articles/10.3389/frobt.2025.1518965/full

[37] vLLM Team. 2024. Inside vLLM: Anatomy of a High-Throughput LLM Inference System. https://blog.vllm.ai/2025/09/05/anatomy-of-vllm.html.

[38] Xin Wang, Yu Zheng, Zhongwei Wan, and Mi Zhang. 2025. SVD-LLM: Truncation-aware Singular Value Decomposition for Large Language Model Compression. In *International Conference on Learning Representations*.

[39] Xin Wang, Yu Zheng, Zhongwei Wan, and Mi Zhang. 2025. SVD-LLM: Truncation-aware Singular Value Decomposition for Large Language Model Compression. In *ICLR*.

[40] Xiaoxia Wu, Zhewei Yao, and Yuxiong He. 2023. Understanding INT4 Quantization for Language Models: Latency Speedup, Composability, and Failure Cases. *arXiv preprint arXiv:2301.12017* (2023).

[41] Jinqi Xiao, Chengming Zhang, Yu Gong, Miao Yin, Yang Sui, Lizhi Xiang, Dingwen Tao, and Bo Yuan. 2023. HALOC: Hardware-Aware Automatic Low-Rank Compression for Compact Neural Networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 10464–10472.

[42] Yiming Zhang, Chenhao Li, Jing Sun, Xin Huang, and Yuxuan Wang. 2024. Dissecting the NVIDIA Hopper Architecture through Microbenchmarking and Multiple Level Analysis. *arXiv preprint arXiv:2501.12084* (2024).

[43] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2023. H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. *arXiv preprint arXiv:2306.14048* (2023).