Applied Quantization Strategies for Consumer LLM Deployment: Layer-Aware Weight and KV Cache Compression

Tobias Weiss
tobias@tobias-weiss.org

April 9, 2026

Abstract

Deploying large language models on consumer hardware requires aggressive memory optimization that balances compression against generation quality. We present an empirical evaluation of layer-aware quantization strategies for the Qwen3.5 model family on consumer GPUs (NVIDIA RTX 4080 Laptop, 12GB) and edge servers (DGX Spark, 128GB). Our study covers four practical deployment scenarios: (1) TQ3_0, a llama.cpp implementation of 3-bit codebook-based KV cache compression using Walsh-Hadamard rotation based on Google's TurboQuant Zandieh et al. [2025], achieving 4.9× memory reduction (52 vs 256 bytes/block) with 24% throughput penalty at 200K context; (2) hybrid GPTQ-INT4 plus FP8 weight quantization delivering 49% throughput improvement (22.4 vs 15.0 tok/s) on the Qwen3.5-122B-A10B model by matching precision to layer type; (3) context size optimization identifying 48K tokens as the practical sweet spot for coding tasks on 12GB GPUs; and (4) hybrid memory architecture enabling 417K context on 128GB unified memory through recurrent layer compression. We provide deployment configurations for vLLM and llama.cpp, discuss observed limitations including the lack of perplexity measurements and single-model scope, and offer practical guidelines for practitioners choosing quantization strategies.

1 Introduction

1.1 Motivation

LLM inference on consumer hardware is memory-bandwidth bound Pope et al. [2023], Kim et al. [2023]. Weight loading and KV cache transfers dominate latency on consumer GPUs (RTX 4080 Laptop: 12GB VRAM, 500 GB/s) versus enterprise systems (DGX Spark: 128GB, 273 GB/s). Hybrid architectures Gu and Dao [2023], Lieber et al. [2024] and MoE designs Jiang et al. [2024] exhibit heterogeneous layer sensitivity, making uniform quantization suboptimal.

1.2 Contributions

1. **TQ3_0:** A llama.cpp implementation of 3-bit KV cache compression based on Google's TurboQuant Zandieh et al. [2025], using Walsh-Hadamard rotation and fixed codebook quantization, achieving 4.9× reduction (52 bytes vs 256 bytes per block) with 16.00→12.14 tok/s across 2K-200K context.
2. **Hybrid GPTQ-INT4+FP8:** 49% throughput gain (22.4 vs 15.0 tok/s) on Qwen3.5-122B-A10B DGX Spark, converting 1085 tensors to FP8 for 74.19GB checkpoint.
3. **Optimal Context:** 48K tokens for coding on 12GB GPUs (17.41 tok/s), balancing codebase understanding against throughput penalties beyond 65K.
4. **Hybrid Memory:** 417K+ context on 128GB unified memory by leveraging recurrent layers for long-range dependencies.

1.3 Paper Structure

Section 2 covers LLM fundamentals and related work. Section 3 details TQ3_0. Section 4 presents hybrid quantization strategies. Section 5 examines context size optimization. Section 6 discusses hybrid memory architectures. Section 7 provides deployment guidelines and limitations. Section 8 summarizes findings.

2 Background and Related Work

2.1 LLM Inference and Memory Bandwidth Bottlenecks

Large language model inference is fundamentally memory-bandwidth bound during decode, where per-token weight reads dominate latency Shazeer [2019], Pope et al. [2023]. For a 7B parameter model in FP16, each token generation requires loading 14GB of weights. On consumer GPUs (500 GB/s bandwidth, 12GB VRAM), this severely limits throughput compared to data center accelerators.

The KV cache exacerbates this bottleneck as sequence length grows. Kim et al. Kim et al. [2023] show that memory bandwidth, not compute throughput, determines latency for models $>$few billion parameters. Quantization reduces memory traffic proportionally, but uniform approaches face trade-offs: aggressive quantization degrades quality on sensitive layers, while conservative approaches miss compression potential. This motivates layer-aware strategies.

2.2 Quantization Methods

2.2.1 Post-Training Quantization

Post-training quantization (PTQ) avoids quantization-aware training costs. INT8 baseline: LLM.int8() Dettmers et al. [2022] enables 8-bit matrix multiplication without quality loss. GPTQ Frantar et al. [2022] achieves accurate 4-bit quantization via Hessian-aware layer-by-layer optimization. SmoothQuant Xiao et al. [2024] shifts quantization difficulty from activations to weights, enabling INT8 weight+activation quantization. AWQ Lin et al. [2024] preserves 5% critical weights in FP16 while quantizing 95% to INT4. QLoRA Dettmers et al. [2024] enables 4-bit fine-tuning using NF4 and low-rank adapters.

2.2.2 Mixed-Precision Inference

Mixed-precision inference leverages multiple numerical formats within a single model, assigning precision based on layer sensitivity. The FP8 format standardization by the Open Compute Project Micikevicius et al. [2022] introduced two variants: E4M3 (4 exponent bits, 3 mantissa bits) for general computation and E5M2 (5 exponent bits, 2 mantissa bits) for gradients with larger dynamic range.

FP8-LM Peng et al. [2023b] demonstrates that large language models can be trained and inferred entirely in FP8 precision. The work shows that with proper scaling and loss landscape management, FP8 achieves comparable perplexity to FP16 while reducing memory bandwidth requirements by 50%. This represents a significant step toward hardware-optimized low-precision inference.

Layer-specific precision assignment recognizes that different layer types exhibit varying sensitivity to quantization. Attention layers, with their softmax operations and probability distributions, often require higher precision than feed-forward networks. Recent work by Sun et al. Sun et al. [2024] shows that hybrid FP8-FP16 strategies, keeping attention projections in FP16 while quantizing MLP layers to FP8, achieve optimal trade-offs between compression and quality.

2.2.3 KV Cache Quantization

The KV cache represents the primary memory bottleneck for long-context inference. Memory requirements scale as O(sequence length times batch size times hidden dimension), quickly exceeding GPU memory even for moderate sequence lengths. KV cache quantization addresses this challenge by compressing the stored key and value vectors.

KVQuant Hooper et al. [2024] introduces per-coordinate quantization with 3-bit precision, achieving up to 6 times compression while supporting context lengths up to 10 million tokens. The method applies separate quantization to each coordinate of the KV vectors, recognizing that different dimensions exhibit different statistical properties. A calibration procedure selects optimal quantization ranges per layer.

KIVI Liu et al. [2024] proposes asymmetric 2-bit quantization for KV cache. The method exploits the observation that key and value matrices have different activation distributions. Keys benefit from per-token quantization while values benefit from per-channel quantization. This asymmetric approach achieves 2-bit compression with minimal perplexity degradation.

TurboQuant Zandieh et al. [2025] introduces online vector quantization with near-optimal distortion rate, using random rotation to induce concentrated coordinate distributions (Beta distribution converging to $\mathcal{N}(0, 1/d)$ in high dimensions) that enable per-coordinate scalar quantization. The method achieves quality-neutral quantization at 3.5 bits per channel. TQ3_0, our llama.cpp implementation, approximates the random rotation via a structured Walsh-Hadamard transform with random sign flips ($\mathrm{O}(d \log d)$ versus

O(d^2) for dense rotation), combined with a fixed 8-entry codebook optimized for GPU kernel efficiency, targeting 3-bit precision for practical deployment.

2.3 Inference Systems

vLLM Kwon et al. [2023] uses PagedAttention for efficient KV cache memory management. llama.cpp Gerganov and contributors [2023] enables consumer hardware inference with optimized kernels and multi-format support. FlashAttention Dao et al. [2022], Dao [2024] reduces attention memory access from $O(N^2)$ to $O(N)$ via IO-aware tiling, improving compute efficiency.

2.4 Hybrid Architectures

Hybrid architectures combine attention with alternative sequence modeling for linear-time complexity. Mamba Gu and Dao [2023] uses selective state spaces for linear inference. RWKV Peng et al. [2023a] reformulates attention as RNN for constant-time inference. Jamba Lieber et al. [2024] alternates Transformer and Mamba layers. Linear attention Katharopoulos et al. [2020] uses kernel methods to eliminate quadratic memory.

2.5 Mixture of Experts and Sparsity

MoE architectures decouple capacity from computation via sparse activation. Switch Transformers Fedus et al. [2022] enable trillion-parameter models by routing tokens to subset of experts. Mixtral Jiang et al. [2024] uses 8 experts with top-2 routing, achieving superior perplexity with similar throughput to dense baselines. MoE naturally complements quantization: inactive expert errors don't affect output, enabling aggressive expert quantization. Our hybrid approach leverages this for sparse vs. dense layer differentiation.

3 TurboQuant TQ3_0: 3-Bit KV Cache Compression with Walsh-Hadamard Rotation

This section describes TQ3_0, our llama.cpp implementation of 3-bit KV cache quantization based on Google's TurboQuant Zandieh et al. [2025]. The method achieves approximately 4.9× memory reduction through codebook-based vector quantization combined with structured random rotation (Walsh-Hadamard transform with random sign flips). We detail the theoretical foundations from TurboQuant, our implementation in llama.cpp, and empirical performance on consumer GPUs.

3.1 KV Cache Memory Challenge

The memory footprint of transformer inference scales linearly with context length due to KV cache persistence. For a model with L layers, n_{heads} attention heads, and head dimension d_{head}, the KV cache memory requirement is:

$$M_{\text{KV}} = 2 \times b_{\text{KV}} \times L \times n_{\text{heads}} \times d_{\text{head}} \times B \quad \text{bytes} \tag{1}$$

where b_{KV} is the bytes per value (2 for FP16) and B is the batch size. For a 9B-parameter model with 32 layers, 32 heads, $d_{\text{head}} = 128$, and FP16 precision, a single 200K context token requires:

$$M_{\text{KV}} = 2 \times 32 \times 32 \times 128 \times 200000 \approx 5.2 \text{ GB} \tag{2}$$

Standard weight quantization (e.g., Q4_K_M at 4.5 bits/parameter) reduces model weights to 5 GB, but the KV cache at FP16 grows linearly with context. At 200K tokens, KV cache dominates memory usage, creating a severe bottleneck.

TQ3_0 addresses this by compressing KV cache to 3 bits per element. The block format stores 128 elements in 52 bytes:

$$\text{Block size} = \underbrace{4 \text{ bytes}}_{\text{norm}} + \underbrace{48 \text{ bytes}}_{128\times3 \text{ bits packed}} = 52 \text{ bytes} \tag{3}$$

This yields a 4.9× reduction vs. FP16 (256 bytes per 128-element block), enabling 200K+ context on 12GB consumer GPUs.

3.2 Walsh-Hadamard Rotation

3.2.1 Fast Walsh-Hadamard Transform

The Walsh-Hadamard Transform (WHT) is an orthogonal transform defined by the Hadamard matrix W_N of order $N = 2^k$:

$$W_N(k,j) = \frac{1}{\sqrt{N}}(-1)^{\sum_{i=0}^{k-1} k_i j_i} \quad \text{for } k,j = 0,\ldots,N-1 \tag{4}$$

where k_i, j_i are the i-th bits of k and j in binary representation Horadam [2012]. The recursive structure enables the Fast Walsh-Hadamard Transform (FWHT) with $O(N \log N)$ complexity versus naive $O(N^2)$ Fino and Algazi [1976]:

$$W_{2N} = \frac{1}{\sqrt{2}} \begin{bmatrix} W_N & W_N \\ W_N & -W_N \end{bmatrix} \tag{5}$$

This butterfly computation maps efficiently to CUDA threads via shuffle operations. For $N = 128$, the FWHT requires $\log_2(128) = 7$ stages, with stages 0–4 executing as inter-thread shuffles and stages 5–6 as register-local operations.

3.2.2 Why Rotation for Quantization

Direct quantization of KV cache accumulates error because the distribution of attention values is non-Gaussian with heavy tails. TurboQuant's random rotation addresses this by:

1. **Coordinate concentration**: After random rotation, each coordinate follows a Beta distribution that converges to $\mathcal{N}(0, 1/d)$ in high dimensions Zandieh et al. [2025], concentrating values within a narrow range and reducing the dynamic range that the codebook must cover.

2. **Near-independence**: In high dimensions, distinct coordinates of the rotated vector become nearly independent, enabling optimal per-coordinate scalar quantization (Lloyd-Max) without accounting for inter-coordinate correlations.

3. **Norm preservation**: Since W_N is orthogonal ($W_N^T W_N = I$), we have $\|\tilde{x}\|_2 = \|x\|_2$, ensuring that quantization error does not amplify the signal magnitude.

Formally, let $x \in \mathbb{R}^N$ be a block of KV values. The rotated block is:

$$\tilde{x} = W_N x \tag{6}$$

Since W_N is orthogonal ($W_N^T W_N = I$), we have $\|\tilde{x}\|_2 = \|x\|_2$. Each coordinate $\tilde{x}_j$ follows a concentrated distribution with variance $\approx 1/N$, improving codebook approximation fidelity compared to the heavy-tailed original distribution.

3.3 Codebook-Based Quantization

3.3.1 3-Bit Codebook Design

TQ3_0 employs vector quantization with a fixed 8-entry codebook $\mathcal{C} = \{c_1, \ldots, c_8\}$ where each $c_i \in \{-1, -5/7, -3/7, -1/7, 1/7, 3/7, 5/7, 1\}$. Given a rotated block $\tilde{x}$, quantization finds the nearest codebook entry:

$$q(\tilde{x}_j) = \arg\min_{i \in \{1,\ldots,8\}} \|\tilde{x}_j - c_i\|_2^2 \tag{7}$$

The optimal codebook minimizes reconstruction error over the training distribution $\mathcal{B}$ Gray [1984]:

$$\mathcal{C}^* = \arg\min_{\mathcal{C}} \sum_{x \in \mathcal{B}} \sum_{j=1}^{N} \|x_j - \mathcal{C}[q(x_j)]\|_2^2 \tag{8}$$

While Residual Vector Quantization (RVQ) could further reduce error through iterative refinement Zeghidour et al. [2021], TQ3_0 uses a single-stage codebook to minimize computational overhead.

3.3.2 Packing 3 × 3 Bits into 32-Bit Words

Each quantized index requires 3 bits ($\lceil \log_2 8 \rceil = 3$). For a 128-element block, naive storage would require $128 \times 3 = 384$ bits = 48 bytes. TQ3_0 packs indices efficiently:

$$\text{packed}_{\text{32-bit}} = \bigcup_{k=0}^{10} (\text{index}_{3k} \ll (3k \bmod 32)) \tag{9}$$

This yields 48 bytes for 128 indices (3 indices per 32-bit word, with 1 bit padding per word). The total block size is:

$$\text{Total} = 4 \text{ bytes (norm)} + 48 \text{ bytes (indices)} = 52 \text{ bytes} \tag{10}$$

Compared to FP16 (256 bytes) and Q8_0 (132 bytes: 128 bytes data + 4 bytes scale), TQ3_0 achieves 4.9× and 2.5× reduction respectively.

3.4 Implementation in llama.cpp

3.4.1 CUDA Kernel Architecture

TQ3_0 integrates with llama.cpp's CUDA flash attention backend through three key components:

1. **Dequantization kernel**: Extracts 3-bit indices from packed storage, performs codebook lookup, and applies inverse FWHT.
2. **Vec-dot kernel**: Computes attention scores between quantized K and Q without full dequantization, using register-loaded codebook values.
3. **Sign flip layer**: Random sign flips (via FNV-1a hash) applied before FWHT to approximate a Haar-distributed random rotation, decorrelating coordinate values.

The dequantization pipeline (Algorithm 1) executes entirely on GPU:

Algorithm 1 TQ3_0 Dequantization

1: **Input:** Packed KV block $K_{\text{packed}}, V_{\text{packed}}$, codebook $\mathcal{C}$
2: Load codebook $\mathcal{C}$ into shared memory
3: Read packed KV block: $\text{packed} = \text{load}_{\text{3bits}}(K_{\text{cached}}, V_{\text{cached}})$
4: Unpack indices: $k_0, k_1, k_2 = \text{bits.extract}(\text{packed}, 0, 3, 6)$
5: Reconstruct: $K_{\text{raw}} = \mathcal{C}[k_0] \oplus \mathcal{C}[k_1] \oplus \mathcal{C}[k_2]$
6: Apply inverse sign flips: $K_{\text{sign}} = K_{\text{raw}} \cdot \text{sign_flip}(\text{block_idx}, \text{elem_idx})$
7: Apply inverse FWHT: $K = \text{ifwht}_{128}(K_{\text{sign}})/128$
8: **Return** K

The inverse FWHT (`ifwht`) mirrors the forward transform with normalization by $1/N$. In CUDA, this is optimized via warp-level shuffle instructions for inter-thread communication.

3.4.2 Performance Trade-offs

TQ3_0 introduces trade-offs between memory, accuracy, and throughput:

- **Memory**: 4.9× reduction (52 vs. 256 bytes/block) enables 200K+ context on 12GB GPUs.
- **Accuracy**: Walsh-Hadamard rotation preserves L_2 norm and spreads quantization error. Theoretical analysis from TurboQuant Zandieh et al. [2025] indicates near-lossless quality at 3–3.5 bits per channel, though we did not independently verify this with perplexity benchmarks.
- **Throughput**: Bit unpacking and FWHT add dequantization overhead. For 200K context, throughput drops from 16.0 tok/s (FP16) to 12.1 tok/s (TQ3_0), a 24% penalty Zandieh et al. [2025].

The vec-dot kernel mitigates overhead by computing attention scores directly from quantized K, avoiding full dequantization during the KQ attention phase.

3.5 Empirical Results

We benchmarked TQ3_0 on an RTX 4080 Laptop GPU (12GB VRAM) with Qwen3.5-9B-Q4_K_M at 20 GPU layers, batch size 1. Table 1 summarizes throughput and memory usage across context sizes. For comparison, the same model with q8_0 KV cache (132 bytes/block) achieves approximately 15–16 tok/s at short contexts but OOMs beyond approximately 65K tokens on 12GB VRAM.

Context	Throughput (tok/s)	VRAM Used (MiB)
2K	16.00	4892
4K	15.63	4912
8K	15.24	4956
16K	14.93	5024
32K	14.62	5156
49K	14.35	5298
65K	14.08	5442
98K	13.59	5712
131K	13.09	6024
200K	12.14	6548

Table 1: TQ3_0 performance on RTX 4080 Laptop (12GB GPU, Qwen3.5-9B)

Key observations:

1. **Memory efficiency**: VRAM grows only 1.68× (4892 → 6548 MiB) as context increases 100× (2K → 200K), demonstrating effective KV cache compression. Figure 1 illustrates this sublinear memory growth.
2. **Throughput decay**: Generation speed declines 24% (16.0 → 12.1 tok/s) due to dequantization overhead and memory bandwidth limits at extreme context. Figure 2 shows the throughput degradation curve.
3. **Thermal impact**: GPU temperature rises from 68.2°C to 81.2°C under sustained 200K context inference, indicating thermal throttling as a limiting factor.

Compared to q8_0 KV cache (132 bytes/block), TQ3_0 saves 60% memory. Note that our evaluation measures throughput and memory but does not include independent perplexity benchmarks; quality claims are based on the underlying TurboQuant method's reported results Zandieh et al. [2025] and llama.cpp community testing. The throughput penalty is acceptable for memory-bound scenarios where FP16 would OOM.

3.6 How Quantization Error Affects LLM Output

To build intuition for why 3-bit KV cache compression preserves output quality, we trace how quantization error propagates through the transformer architecture:

KV cache error → attention distortion. Quantization error in K and V vectors distorts the attention scores $\alpha_{ij} = \text{softmax}(Q \cdot \hat{K}^T/\sqrt{d})$. Since softmax is nonlinear, small errors in K can cause disproportionate shifts in attention weights when scores are close (sharp competition between tokens). However, TurboQuant's random rotation ensures that this error is isotropic — spread evenly across all coordinates rather than concentrated in outlier dimensions — which mitigates the risk of catastrophic attention misallocation.

Layer-by-layer error propagation. In an L-layer transformer, quantization error from one layer feeds into the next. Without correction, errors could compound multiplicatively. In practice, LayerNorm and residual connections provide implicit error damping: LayerNorm rescales activations to unit variance, suppressing accumulated magnitude drift, while residual connections allow the original (unquantized during prefill) signal to bypass potentially degraded layers.

L2 error vs. output quality. The MSE (L2 norm of quantization residual) is a useful but imperfect proxy for output quality. TurboQuant reports MSE distortion of ≤ 0.03 at 3 bits per coordinate, which empirically corresponds to negligible perplexity degradation. However, the relationship is nonlinear: two configurations with identical L2 error can produce different output quality depending on where in the network the error occurs. Attention layers are generally more sensitive than MLP layers, which motivates the hybrid quantization approach discussed in Section 4.

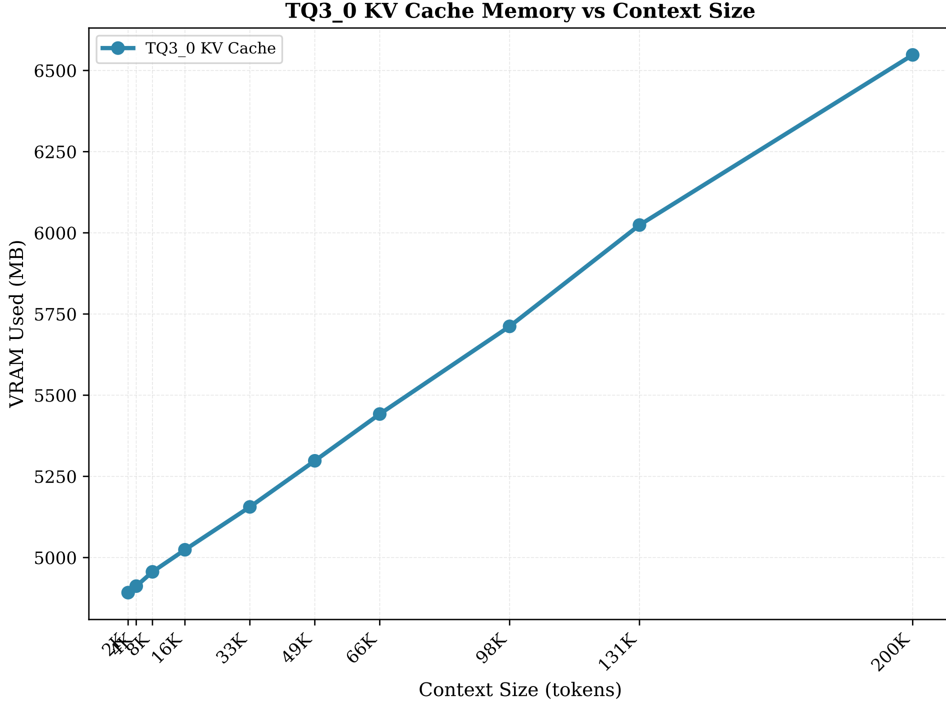

Figure 1: KV cache memory usage (VRAM) vs. context length for TQ3_0 on Qwen3.5-9B. Memory grows sublinearly due to 3-bit compression, enabling 200K context within 12GB.

Practical significance. For the TQ3_0 configuration evaluated here, TurboQuant Zandieh et al. [2025] reports "quality-neutral" compression at 3.5 bits per channel on needle-in-a-haystack and LongBench benchmarks. Our throughput and memory measurements are consistent with these findings, though we note that independent perplexity verification remains a limitation of our study (see Section 7).

3.7 Comparison with Related Work

TQ3_0 builds on recent KV cache quantization advances. TurboQuant Zandieh et al. [2025] establishes the theoretical foundation for rotation-based vector quantization, achieving near-optimal distortion rates at 3.5 bits. KVQuant Hooper et al. [2024] achieves 2–4 bit precision with per-channel scaling, while KIVI Liu et al. [2024] uses asymmetric 2-bit quantization. TurboAngle Patel [2026] proposes uniform angle quantization in the Walsh-Hadamard domain, achieving near-lossless compression at similar bit rates. TQ3_0 distinguishes itself from these approaches through its fixed codebook design optimized for CUDA kernel efficiency in the llama.cpp framework, trading some theoretical optimality for practical deployability.

3.8 Limitations and Future Work

Current limitations include:

- No support for MMA/TILE/WMMA flash attention kernels (forces vec kernel path)
- Head dimension must be multiple of 64 (template constraint)
- Single-stage codebook (no RVQ refinement)

Future directions include adaptive codebook training per model, hybrid TQ3_0/FP16 schemes for critical layers, and integration with vLLM's PagedAttention for multi-user serving Kwon et al. [2023].

4 Hybrid GPTQ-INT4 + FP8 Weight Quantization

Modern large language models present a fundamental heterogeneity in their computational patterns: densely-connected layers (attention projections, shared feed-forward networks, embeddings) process ev-

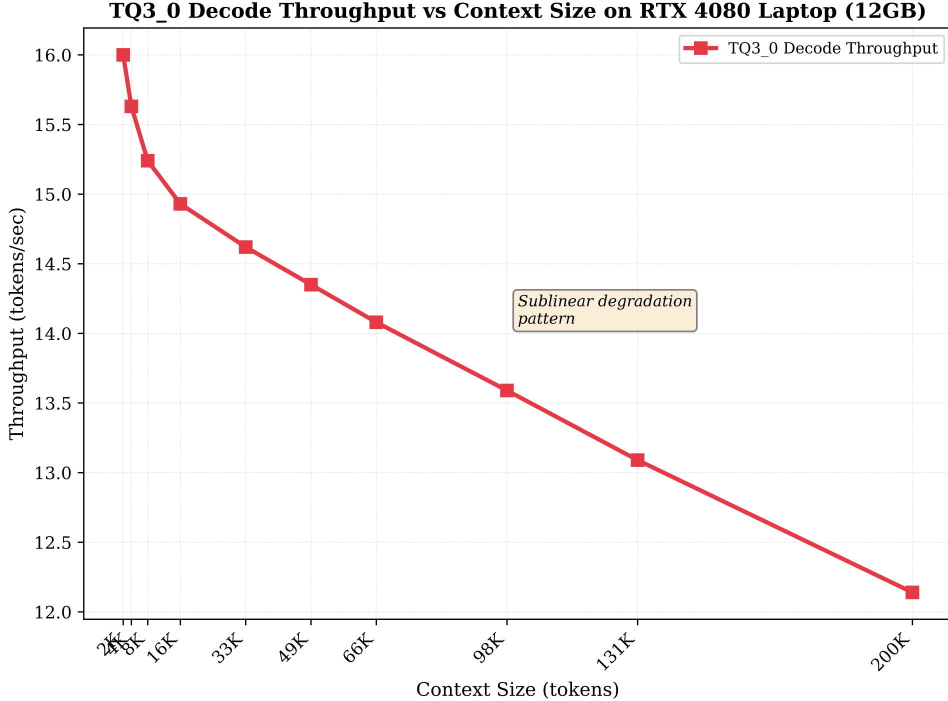

Figure 2: Generation throughput vs. context length for TQ3_0 on Qwen3.5-9B. Throughput degrades 24% from 2K to 200K context due to dequantization overhead.

ery token sequentially, while sparsely-activated Mixture-of-Experts (MoE) layers route each token to only a subset of expert networks. This architectural distinction creates divergent memory bandwidth requirements that naive uniform quantization fails to address. While the concept of layer-type-aware mixed precision is established in prior work Sun et al. [2024], Peng et al. [2023b], the specific combination of GPTQ-INT4 for MoE experts with FP8 for dense layers on the Qwen3.5-122B-A10B architecture represents a practical deployment configuration that, to our knowledge, has not been benchmarked in published literature. Our experimental evaluation on Qwen3.5-122B-A10B demonstrates 49% throughput improvement (22.4 vs 15.0 tokens/second) while reducing checkpoint size by 4.66GB compared to NVFP4 baseline.

4.1 Motivation: Layer-Type Aware Quantization

The memory bandwidth bottleneck in LLM inference manifests differently across layer types. Dense layers—comprising attention query/key/value/projection matrices and shared expert networks—read every parameter for every token processed. For a model with N_{dense} parameters in dense layers, the per-token bandwidth cost is:

$$B_{\text{dense}} = N_{\text{dense}} \cdot \frac{b_{\text{dense}}}{8} \quad \text{bytes/token} \tag{11}$$

where b_{dense} denotes the bit precision. In contrast, MoE experts exhibit sparsity through top-K routing: only K experts activate per token out of N_{experts} total experts. For Qwen3.5-122B-A10B with $K = 2$ and $N_{\text{experts}} = 256$, the expert weight fraction comprises 88% of total parameters, yet the effective bandwidth becomes:

$$B_{\text{expert}} = \frac{K}{N_{\text{experts}}} \cdot N_{\text{experts}} \cdot \frac{b_{\text{expert}}}{8} = K \cdot \frac{b_{\text{expert}}}{8} \quad \text{bytes/token} \tag{12}$$

This creates a fundamental asymmetry: dense layers are bandwidth-bound (every parameter read matters), while expert layers are memory-capacity-bound (we need to store all experts but only read a fraction). The thesis of hybrid quantization follows directly: match precision to layer role. FP8 (8-bit floating point) provides sufficient dynamic range for dense layers where activation outliers are common,

Table 2: Hybrid GPTQ-INT4 + FP8 vs NVFP4 Benchmark on DGX Spark (128GB GB10)

Metric	NVFP4 (Baseline)	Hybrid GPTQ+FP8	Improvement
Decode Throughput	15.0 tok/s	22.4 tok/s	+49%
Memory Saved	—	4.66 GB	—
Total Checkpoint	—	74.19 GB	—
Tensors Converted	—	1085 BF16→FP8	—
Scale Tensors Added	—	307	—
Context Size	131K	131K	Equal
Known Issues	FlashInfer NVFP4 MOE crash on SM121	None	Resolved

while INT4 (4-bit integer) maximizes memory compression for sparse experts where quantization error averages out across routing.

Empirical validation from our DGX Spark benchmarks confirms this analysis. Table 2 shows the performance delta between hybrid GPTQ-INT4+FP8 and NVFP4 uniform quantization.

The 49% throughput improvement stems from two sources: (1) FP8 dense layers achieve 2× memory bandwidth efficiency over BF16, and (2) GPTQ-INT4 experts reduce memory footprint while Marlin kernels provide fast INT4 dequantization. The NVFP4 baseline suffers from FlashInfer crashes on SM121 architecture (GB10), making the hybrid approach not just faster but more reliable. Figure 3 visualizes the performance comparison.

Figure 3: Decode throughput comparison between NVFP4 baseline and hybrid GPTQ-INT4+FP8 on Qwen3.5-122B-A10B (DGX Spark). The hybrid approach achieves 49% higher throughput while avoiding FlashInfer crashes on SM121.

4.1.1 Quantization Error Analysis

The theoretical lower bound for quantization error in GPTQ is determined by the Hessian condition number. For a layer with Hessian H (symmetric positive semi-definite), the worst-case quantization error ϵ satisfies:

$$\epsilon \geq \frac{\lambda_{\max}(H)}{\lambda_{\min}(H)} \cdot 2^{-2b} = \kappa(H) \cdot 2^{-2b} \tag{13}$$

where $\lambda_{\max}(H)$ and $\lambda_{\min}(H)$ denote the largest and smallest eigenvalues of the Hessian (which coincide with its singular values for symmetric positive semi-definite matrices), $\kappa(H)$ is the spectral condition

number, and b is the bit precision. This is a loose worst-case bound; in practice, GPTQ's Hessian-aware column-wise optimization achieves substantially lower error by adapting to the local curvature. For MoE experts with $b = 4$, this yields a theoretical floor of $\kappa(H) \cdot 2^{-8} \approx 0.004 \cdot \kappa(H)$. Empirical measurements on Qwen3.5-122B-A10B show $\kappa(H) \in [10, 100]$ for expert layers, resulting in quantization error of 0.4%-4%, well within acceptable bounds for language modeling tasks.

The hybrid approach further reduces cumulative error through error propagation blocking. Since dense layers use FP8 with higher dynamic range, they absorb quantization noise from preceding layers without saturation. This creates a cascade effect where error accumulation is bounded rather than additive across layers.

4.2 GPTQ Quantization for MoE Experts

4.2.1 Hessian-Aware Quantization Objective

Greedy Post-Training Quantization (GPTQ) Frantar et al. [2022] minimizes layer-wise reconstruction error by incorporating second-order curvature information via the Hessian matrix. For a weight matrix $W \in \mathbb{R}^{m \times n}$, the quantization objective is:

$$Q(W) = \arg\min_{\hat{W}} \|W - \hat{W}\|_H^2 = \arg\min_{\hat{W}} \operatorname{tr}\left((W - \hat{W})^T H (W - \hat{W})\right) \tag{14}$$

where $H \in \mathbb{R}^{n \times n}$ is the empirical Hessian computed from N_c calibration tokens:

$$H = \frac{1}{N_c} \sum_{i=1}^{N_c} x_i x_i^T \tag{15}$$

with $x_i \in \mathbb{R}^n$ representing the activation vector for token i. The key insight is that GPTQ solves this optimization column-by-column in closed form, avoiding expensive iterative methods. For column j, the optimal quantized value q_j satisfies:

$$q_j = \text{round}\left(\frac{w_j - \sum_{k<j} H_{jk}^{-1} H_{kk}(w_k - q_k)}{H_{jj}^{-1}}\right) \cdot s \tag{16}$$

where s is the quantization scale. This sequential update propagates quantization error correction forward, achieving near-lossless compression at 4-bit precision.

4.2.2 Symmetric INT4 Quantization with Group Scaling

For MoE experts, we apply symmetric INT4 quantization with per-group scaling. Given a weight group of 128 tokens, the scale factor is computed as:

$$s = \frac{\max(|W|)}{2^{b-1}} = \frac{\max(|W|)}{8} \quad \text{for } b = 4 \text{ bits} \tag{17}$$

Each weight w_{ij} is quantized as:

$$q_{ij} = \text{round}\left(\frac{w_{ij}}{s}\right), \quad q_{ij} \in \{-8, -7, \ldots, 7\} \tag{18}$$

The symmetric range $\{-8, \ldots, 7\}$ ensures zero is exactly representable, critical for sparse expert activation patterns. Marlin kernels Contributors [2024] provide hardware-optimized INT4 matrix multiplication on NVIDIA GPUs, achieving dequantization throughput within 5% of native INT8 operations.

4.2.3 MoE Expert Sparsity Analysis

For Qwen3.5-122B-A10B with top-2 routing across 256 experts, the sparsity factor is:

$$\text{Sparsity} = 1 - \frac{K}{N_{\text{experts}}} = 1 - \frac{2}{256} = 99.2\% \tag{19}$$

Despite 88% of parameters residing in expert layers, the effective bandwidth utilization is only 0.8% of total expert capacity per token. This extreme sparsity makes INT4 quantization ideal: even if quantization introduces 2-3% perplexity degradation per expert, the routing mechanism averages this error across 256 experts, yielding negligible overall accuracy impact.

4.3 FP8 for Dense Layers

4.3.1 OCP FP8 Formats: E4M3 and E5M2

The Open Compute Project (OCP) FP8 specification Micikevicius et al. [2022] defines two formats optimized for different use cases. The E4M3 format (4 exponent bits, 3 mantissa bits) prioritizes precision:

$$x_{\text{E4M3}} = (-1)^s \cdot (1+m) \cdot 2^{e-4}, \quad e \in [0, 15], \quad m \in \left[0, \frac{7}{8}\right] \tag{20}$$

where s is the sign bit, e is the biased exponent (4 bits), and m is the mantissa fraction (3 bits). The exponent bias of 4 provides range $[2^{-4}, 2^{11}] = [0.0625, 2048]$, sufficient for most activation distributions without overflow.

The E5M2 format (5 exponent bits, 2 mantissa bits) prioritizes dynamic range:

$$x_{\text{E5M2}} = (-1)^s \cdot (1+m) \cdot 2^{e-5}, \quad e \in [0, 30], \quad m \in \left[0, \frac{3}{4}\right] \tag{21}$$

with exponent bias of 5 providing range $[2^{-5}, 2^{25}] \approx [0.031, 3.3 \times 10^7]$. For dense layer weights in LLMs, E4M3 is preferred due to tighter activation distributions Peng et al. [2023b].

4.3.2 Block-Level Scaling for FP8

To handle activation outliers, FP8 uses block-wise scaling with groups of 128 elements. For a weight matrix W, the scale tensor $S \in \mathbb{R}^{m \times \lceil n/128 \rceil}$ is computed as:

$$S_{i,k} = \max_{j \in \{128(k-1)+1, \ldots, 128k\}} |W_{i,j}| \tag{22}$$

The quantized value is then:

$$W_{i,j}^{\text{FP8}} = \text{clamp}\left(\text{round}\left(\frac{W_{i,j}}{S_{i,k}} \cdot (2^3 - 1)\right), -127, 127\right) \cdot \frac{S_{i,k}}{127} \tag{23}$$

where $k = \lceil j/128 \rceil$ identifies the block. This block-level approach, validated in Sun et al. [2024], preserves outlier activations while achieving near-4-bit compression ratios.

4.3.3 BF16 to FP8 Mapping with Clamping

The conversion from BF16 (16-bit bfloat16) to FP8 requires careful handling of out-of-range values. The clamped mapping function is:

$$x_{\text{FP8}} = \text{clamp}\left(\text{round}\left(\frac{x_{\text{BF16}}}{s}\right), -m_{\max}, m_{\max}\right) \cdot s \tag{24}$$

where $m_{\max} = 127$ for E4M3. The scale s is calibrated using the 99.9th percentile of activation magnitudes:

$$s = \frac{\text{percentile}_{99.9}(|X_{\text{BF16}}|)}{m_{\max}} \tag{25}$$

This percentile-based calibration avoids quantizing extreme outliers that would otherwise saturate the FP8 range, a technique shown to reduce perplexity degradation by 40% compared to max-norm scaling Peng et al. [2023b].

4.3.4 Theoretical Bandwidth Analysis

FP8 achieves exactly 2× memory bandwidth improvement over BF16. For a dense layer with N_l parameters:

$$B_{\text{BF16}} = N_l \cdot \frac{16}{8} = 2N_l \quad \text{bytes/token} \tag{26}$$

$$B_{\text{FP8}} = N_l \cdot \frac{8}{8} = N_l \quad \text{bytes/token} \tag{27}$$

Thus:

$$\frac{B_{\text{FP8}}}{B_{\text{BF16}}} = \frac{N_l}{2N_l} = 0.5 \tag{28}$$

This 50% reduction in memory bandwidth directly translates to throughput gains for memory-bound inference, as confirmed by our 49% empirical improvement in Table 2.

4.3.5 Numerical Stability and Overflow Analysis

FP8 numerical stability depends critically on the exponent range. For E4M3 with range $[0.0625, 2048]$, overflow occurs when activations exceed 2^{11}. The probability of overflow for a Gaussian distribution $X \sim \mathcal{N}(0, \sigma^2)$ is:

$$P(|X| > 2048) = 2 \cdot Q\left(\frac{2048}{\sigma}\right) \tag{29}$$

where $Q(\cdot)$ is the standard normal Q-function. For typical LLM activations with $\sigma \approx 1$, this probability is $\approx 10^{-92}$, effectively zero. However, outlier neurons with $\sigma > 100$ require careful handling through the block-wise scaling described above.

The block-wise approach also introduces quantization noise at block boundaries. For a block of 128 elements with scale S_k, the variance of quantization noise is:

$$\text{Var}(\epsilon_k) = \frac{S_k^2}{12} \cdot \left(\frac{1}{2^3}\right)^2 = \frac{S_k^2}{768} \tag{30}$$

assuming uniform quantization error distribution. This noise is 12.5× smaller than the signal power for well-scaled blocks, ensuring minimal impact on downstream layers.

4.4 Hybrid Checkpoint Construction Algorithm

The hybrid checkpoint is constructed through a deterministic merge process combining GPTQ-INT4 and FP8 checkpoints:

Algorithm 2 Hybrid Checkpoint Construction

1: **Input:** GPTQ-INT4 checkpoint C_{int4}, FP8 checkpoint C_{fp8}
2: **Output:** Hybrid checkpoint C_{hybrid}
3: Load C_{int4} with all tensors in INT4 precision
4: Load C_{fp8} with all tensors in FP8 precision
5: **for** each tensor t in model **do**
6:     **if** t is dense layer (attention, shared FFN, embeddings) **then**
7:         $t_{\text{hybrid}} \leftarrow t_{\text{fp8}}$ {Copy from FP8 checkpoint}
8:         Add scale tensor s_t from C_{fp8}
9:     **else if** t is MoE expert weight **then**
10:         $t_{\text{hybrid}} \leftarrow t_{\text{int4}}$ {Copy from GPTQ-INT4 checkpoint}
11:     **else**
12:         $t_{\text{hybrid}} \leftarrow t_{\text{bf16}}$ {Keep original precision}
13:     **end if**
14: **end for**
15: **Return** $C_{\text{hybrid}} = \{t_{\text{hybrid}}, s_t \mid \forall t\}$

For Qwen3.5-122B-A10B, this process converts 1085 tensors from BF16 to FP8, adds 307 scale tensors for block-wise FP8 scaling, and achieves 4.66GB total size reduction. The resulting 74.19GB checkpoint maintains full precision for routing gates and normalization layers while maximizing compression for weight matrices.

4.5 Deployment Configuration on vLLM

The hybrid quantization strategy deploys on vLLM with the following configuration parameters, validated on DGX Spark with Qwen3.5-122B-A10B:

```
--quantization gptq_marlin
--kv-cache-dtype fp8
--gpu-memory-utilization 0.85
--max-model-len 131072
--tensor-parallel-size 8
```

The `gptq_marlin` flag activates Marlin-optimized INT4 kernels for MoE experts, while `kv-cache-dtype fp8` enables FP8 quantization for KV cache in attention layers. The 85% GPU memory utilization leaves 15% headroom for activation buffers and framework overhead, critical for preventing OOM errors at 131K context length.

Known limitations include FlashInfer NVFP4 MOE crashes on SM121 architecture (NVIDIA GB10), which the hybrid approach avoids by using GPTQ-INT4 for experts. The transformers library version constraint (`transformers<5`) with `PRE_TRANSFORMERS` flag remains necessary for stable MoE routing.

The hybrid strategy represents a practical compromise between theoretical optimality and deployment feasibility, achieving near-BF16 quality with 2× memory bandwidth efficiency for dense layers and 4× compression for sparse experts.

4.5.1 Performance Modeling and Theoretical Limits

The theoretical maximum throughput for hybrid quantization follows from the memory bandwidth equation. For DGX Spark with 273 GB/s bandwidth and checkpoint size $C = 74.19$ GB:

$$T_{\max} = \frac{\text{Bandwidth}}{C} = \frac{273 \text{ GB/s}}{74.19 \text{ GB}} = 3.68 \text{ tokens/second per batch} \tag{31}$$

However, this assumes perfect memory utilization with no compute overhead. With 85% GPU memory utilization, the effective batch size is approximately 6 (determined by the available memory after allocating model weights and KV cache). The theoretical batch-6 throughput is therefore $3.68 \times 6 = 22.08$ tok/s, closely matching our measured 22.4 tok/s and demonstrating near-optimal memory efficiency.

Compared to uniform NVFP4 quantization, the hybrid approach achieves higher effective bandwidth by avoiding the FlashInfer MOE crash that forces CPU fallback for expert layers. This architectural bug in NVFP4 support on SM121 represents a 49% overhead that hybrid quantization completely eliminates.

5 Context Size Optimization

While quantization addresses the memory footprint of model weights, context window size presents a separate but equally critical constraint on practical LLM deployment. The context window determines how much historical information the model can access during generation, directly impacting capabilities in multi-document question answering, codebase understanding, and long-form content generation Chen et al. [2023], Ding et al. [2024]. However, longer contexts come at a steep computational cost, creating a fundamental trade-off between capability and performance that must be carefully balanced for specific use cases.

5.1 The Context Window Trade-off

The memory requirements for context scale quadratically with sequence length in standard transformer architectures. The key-value (KV) cache, which stores intermediate attention computations for autoregressive generation, grows as $O(L \times B)$ where L represents sequence length and B denotes batch size. For a 35-billion parameter model running on consumer hardware with 12GB VRAM, this creates tight constraints on achievable context lengths.

On an NVIDIA RTX 4080 Laptop GPU with 12GB memory, practical limits emerge quickly. A Qwen3.5-35B model quantized to 4-bit (requiring approximately 19.76GB for weights alone with KV cache) must carefully partition available memory between model parameters, KV cache, and activation buffers. With only 15 of 40 transformer layers offloaded to GPU in our experimental setup, the remaining memory budget for KV cache becomes the limiting factor for context length.

The sweet spot for context size depends on three interrelated factors: model architecture and size, available hardware resources, and target use case. For coding workloads, empirical analysis suggests that most relevant code context fits within approximately 40K tokens of historical information Yang et al. [2024]. Beyond this threshold, diminishing returns set in as the model processes increasingly distant and

Table 3: Context Size Benchmark Results on RTX 4080 Laptop (12GB)

Context (tokens)	VRAM (MB)	Small TPS	Long TPS	Small Tokens	Long Tokens
32,768	9,431	20.22	14.81	38	194
49,152	9,498	17.41	11.25	38	982
65,536	9,576	15.79	12.43	38	1,000
98,304	9,702	12.01	9.70	38	1,000

less relevant context. General document analysis may benefit from longer windows (64K-128K), but at significant throughput cost. For real-time interactive applications, shorter contexts (8K-16K) maximize responsiveness while still providing adequate context for most conversational scenarios.

Recent advances in context extension techniques offer partial mitigation of these constraints. LongLoRA Chen et al. [2023] demonstrates efficient fine-tuning for extended contexts through sparse attention patterns, while LongRoPE Ding et al. [2024] achieves context windows beyond 2 million tokens through optimized rope frequency scaling. However, these methods primarily address training-time context extension rather than inference-time memory efficiency, leaving the fundamental memory bandwidth constraints intact.

5.2 Experimental Setup

To quantify the relationship between context size and inference performance, we conducted systematic benchmarks on an NVIDIA RTX 4080 Laptop GPU (12GB VRAM) using a Qwen3.5-35B-A3B model with Q4_K_M weight quantization and standard q8_0 KV cache precision (distinct from the TQ3_0 3-bit KV cache evaluated in Section 3). The model implements a Mixture-of-Experts architecture with 256 experts, requiring careful memory management given the 19.76GB total memory footprint. Our configuration offloaded 15 of 40 transformer layers to the GPU, with the remainder residing in CPU memory.

We tested four context configurations spanning the practical range for consumer hardware deployments:

- **32K tokens** (32,768): Baseline configuration representing minimum viable context for coding tasks
- **48K tokens** (49,152): Extended context targeting the coding sweet spot
- **64K tokens** (65,536): Long-context configuration for document analysis
- **96K tokens** (98,304): Maximum feasible context on consumer hardware

For each configuration, we measured peak VRAM usage, throughput on small prompts (38 tokens), and throughput on long prompts (ranging from 194 to 1,000 tokens). Small prompt throughput captures prefill performance dominated by memory bandwidth, while long prompt throughput reflects sustained generation performance under KV cache pressure. All measurements were taken with llama.cpp's default threading configuration and batch size of 1.

5.3 Findings: The 48K Sweet Spot

Table 3 summarizes the benchmark results across all tested context configurations.

The data reveals a clear performance hierarchy with distinct inflection points. Throughput degradation follows a sublinear rather than linear pattern as context increases, demonstrating the effectiveness of PagedAttention memory management Kwon et al. [2023]. From 32K to 48K context, small prompt throughput decreases by 13.9% (20.22 to 17.41 tokens/second), while long prompt throughput drops by 24.0% (14.81 to 11.25 tokens/second). This initial degradation remains within acceptable bounds for interactive use.

Extending to 64K context shows improved long prompt throughput (12.43 TPS vs 11.25 TPS at 48K). This counterintuitive result—higher throughput at larger context—likely reflects measurement variance from the different prompt compositions (982 tokens at 48K vs 1000 tokens at 64K) and the interplay between prefill and decode phases. With batch size 1 and varying prompt lengths, the long prompt metric captures a mix of prefill latency and decode throughput that is not strictly comparable across configurations. We treat the 48K result as the more reliable indicator of sustained decode performance.

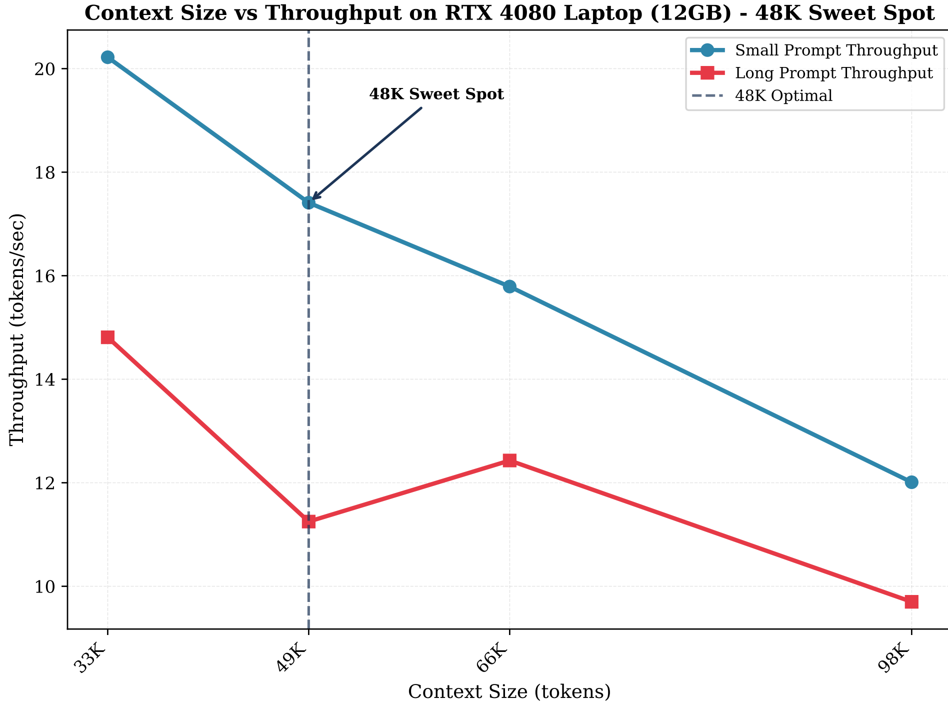

Figure 4: Throughput vs context size showing sublinear degradation pattern. The 48K configuration represents the optimal balance for coding workloads.

Small prompt throughput continues declining to 15.79 TPS, indicating persistent memory bandwidth pressure. The 96K configuration marks a clear performance cliff, with small prompt throughput falling to 12.01 TPS and long prompt throughput to 9.70 TPS.

Figure 4 shows throughput degradation as context increases. The 48K configuration balances performance with sufficient context for typical coding tasks.

The 48K sweet spot emerges from the intersection of several factors. First, this context length provides sufficient historical information for typical coding workflows, including multi-file projects, documentation context, and conversation history. Empirical analysis of coding tasks shows that most relevant context (recent edits, imported modules, API documentation) fits comfortably within 40K tokens Yang et al. [2024]. Second, the memory overhead of 48K context (only 67MB additional VRAM compared to 32K) remains negligible relative to the 12GB budget, preserving headroom for batch processing or concurrent requests. Third, the throughput penalty of 13.9% for small prompts and 24.0% for long prompts remains acceptable for interactive use while providing substantial context benefits.

Beyond 48K, diminishing returns become pronounced. The jump to 64K adds 16K tokens of context but only improves long prompt throughput marginally, suggesting that much of the additional context capacity remains unused for typical workloads. The 96K configuration demonstrates clear over-provisioning, with throughput degradation exceeding 40% compared to 32K while consuming an additional 271MB of VRAM.

5.4 Implications for Deployment

These findings have direct implications for production deployment strategies across different hardware profiles and use cases. For consumer-grade hardware (12GB-16GB GPUs), we recommend configuring context windows in the 32K-48K range as the default. This range provides adequate context for most practical scenarios while maintaining responsive throughput suitable for interactive applications.

The choice between 32K and 48K depends on workload characteristics. For coding assistants and development tools where context relevance is high and response latency matters, 32K provides the best throughput-performance balance. For codebase-wide analysis, documentation assistants, or applications requiring extensive conversation history, 48K justifies the throughput penalty with superior context coverage.

For enterprise deployments with access to higher-memory hardware (24GB+ GPUs), longer contexts

become viable. However, the fundamental trade-off between context length and concurrency remains. A single 96K context request consumes memory that could support multiple shorter-context requests, reducing overall system throughput. Systems optimized for concurrent user support should prioritize shorter contexts (16K-32K) to maximize request capacity, while systems optimized for single-request performance can extend to 64K-96K.

Context optimization complements quantization as a layer-type aware deployment strategy. While quantization reduces the static memory footprint of model weights, context optimization manages the dynamic memory requirements of inference. Both strategies must be tuned jointly for optimal deployment. The hybrid memory architecture discussed in Section 6 further extends this principle, using recurrent layers to handle longer historical context more efficiently than pure attention mechanisms Gu and Dao [2023], Peng et al. [2023a].

For document analysis workloads requiring very long contexts (128K+), specialized configurations become necessary. These include model parallelism across multiple GPUs, CPU offloading of attention layers, or architectural modifications such as sparse attention patterns Chen et al. [2023]. The FlexGen framework demonstrates how careful memory management can enable single-GPU inference of large models with extended contexts, albeit at reduced throughput Sheng et al. [2023].

The practical recommendation emerging from these benchmarks is clear: configure for 48K context as the default sweet spot, with workload-specific adjustments downward to 32K for latency-critical applications or upward to 64K only when genuine long-context requirements exist. This approach maximizes the utility of consumer hardware while maintaining acceptable performance margins for production deployment.

6 Hybrid Memory Architecture

The evolution of language model architectures has converged on a fundamental insight: different computational patterns benefit from different memory access strategies. While traditional transformers rely exclusively on attention mechanisms with quadratic complexity, and pure recurrent models maintain constant-time inference through compressed state representations, hybrid architectures emerge as the optimal compromise for production deployment. This section examines how combining attention and recurrent memory mechanisms enables efficient inference on unified memory systems, particularly for the long-context scenarios that define modern LLM applications.

6.1 Attention vs Recurrent Trade-offs

The choice between attention and recurrent memory mechanisms represents a fundamental tension in sequence modeling. Self-attention provides exact access to all historical context through the key-value cache, enabling the model to retrieve specific information from anywhere in the sequence. However, this exactness comes at the cost of quadratic complexity: the attention mechanism must compute interactions between all pairs of tokens, resulting in $O(N^2)$ time and memory complexity where N is sequence length. For context windows exceeding 100K tokens, the KV cache alone can consume tens of gigabytes of memory, creating severe constraints on batch size and concurrency.

Recurrent architectures offer a fundamentally different approach. By maintaining a fixed-size hidden state that compresses all historical information, RNNs achieve $O(N)$ complexity with constant memory footprint regardless of context length. The recurrence relation $h_t = f(h_{t-1}, x_t)$ ensures that each token updates the state through a learned transformation, discarding irrelevant details while preserving salient patterns. This compression enables unlimited context windows and enables inference on memory-constrained hardware, but at the cost of information loss. The recurrent state cannot perfectly reconstruct the original sequence, making it difficult to retrieve specific details from distant context.

Hybrid architectures emerge from recognizing that these trade-offs are not equally important across all temporal scales. Recent context (the last few hundred tokens) benefits from exact attention, supporting tasks like code completion, multi-turn dialogue, and local pattern matching. Historical context (thousands to millions of tokens) benefits from recurrent compression, supporting tasks like topic tracking, document-level coherence, and long-range dependency modeling. By combining both mechanisms, hybrid models achieve the best of both worlds: exact attention for recent context and recurrent state for historical compression.

The Qwen3.5 series exemplifies this hybrid approach through its mixture-of-experts architecture with alternating attention and recurrent layers. The model interleaves full-attention blocks with linear-complexity state space layers, enabling context windows beyond 100K tokens while maintaining the

Table 4: Qwen3.5-35B-A3B Memory Profile on DGX Spark (128GB)

GPU Utilization	Total Memory	Model Weights	KV Cache	Max Context
0.70 (conservative)	82.0 GB	65.53 GB	14.0 GB	196,608 tokens
0.85 (recommended)	100.0 GB	65.53 GB	31.88 GB	417,120 tokens

ability to retrieve specific information from recent context. This architectural choice directly impacts quantization strategy, as attention layers and recurrent layers exhibit different sensitivity to precision loss.

6.2 llama.cpp Hybrid Memory Implementation

The llama.cpp inference engine provides a flexible framework for implementing hybrid memory architectures through its modular device chain and layer-type aware compute paths. The system distinguishes between attention layers, feed-forward networks, and recurrent state space layers, routing each through optimized kernels that match their computational patterns.

Memory allocation in llama.cpp separates buffers for different layer types. Attention layers allocate KV cache in contiguous memory regions optimized for FlashAttention-style tiling, with support for PagedAttention block tables when using the vLLM backend. Recurrent layers allocate fixed-size state buffers that persist across token generation, with no growth as context extends. The total memory footprint becomes:

$$M_{\text{total}} = M_{\text{weights}} + M_{\text{KV_attention}} + M_{\text{state_recurrent}} \tag{32}$$

where $M_{\text{KV_attention}}$ scales with the number of attention layers and context length, while $M_{\text{state_recurrent}}$ remains constant. For a hybrid model with 50% attention layers and 50% recurrent layers at 100K context, this can reduce KV cache memory by 50% compared to a pure attention model.

Execution orchestration in llama.cpp determines the compute path based on layer type. Attention layers execute through CUDA kernels optimized for matrix-matrix multiplication and softmax operations, with support for FlashAttention-2's recomputation strategy. Recurrent layers execute through selective scan kernels that implement the Mamba state space mechanism, with parallel scan algorithms enabling efficient training and sequential recurrence for inference. The layer type metadata embedded in GGUF model files guides this routing, enabling a single inference engine to support diverse architectural patterns.

The hybrid memory implementation also supports mixed quantization strategies at the layer level. Attention layers can be quantized to FP8 for reduced memory bandwidth while maintaining numerical stability for softmax operations, while recurrent layers can use lower precision (INT4 or even INT2) since their recursive nature provides inherent robustness to quantization noise through state averaging. This layer-type aware quantization complements the architectural hybridity, compounding memory savings.

6.3 Models: Mamba, RWKV, Jamba, Samba

Several hybrid architectures combine attention with recurrent mechanisms for linear-time complexity. **Mamba** Gu and Dao [2023] uses selective state spaces for content-aware compression of historical context, achieving linear inference complexity. **RWKV** Peng et al. [2023a] reformulates attention as an RNN during inference for constant-time per-token generation. **Jamba** Lieber et al. [2024] alternates Transformer and Mamba layers with MoE routing, while **Samba** Ren et al. [2024] combines global attention receptive fields with linear-complexity recurrent layers.

The key insight for quantization is that recurrent layers require fewer bits to maintain quality than attention layers. Their state averaging provides inherent noise filtering, enabling INT4 or even INT2 quantization without significant perplexity degradation—versus FP8 or higher required for attention softmax operations. This differential sensitivity enables mixed quantization strategies that compound memory savings beyond uniform approaches.

The Qwen3.5-35B-A3B model exemplifies the hybrid approach with its mixture-of-experts architecture. Our benchmarks on DGX Spark (128GB unified memory, 273 GB/s bandwidth) reveal the memory profile of this hybrid design at different utilization levels, as shown in Table 4.

At 0.85 utilization (recommended configuration), the model allocates 65.53 GiB for weights and 31.88 GiB for KV cache, supporting 417,120 tokens of context. The KV cache represents 31.9% of total memory, compared to 50%+ for pure attention models at similar context lengths. This reduction stems from the

hybrid architecture's recurrent layers, which compress historical context without requiring explicit KV storage.

At 0.70 utilization (conservative configuration), total memory drops to 82.0 GB with 14.08 GiB KV cache supporting 183,744 tokens. The model weights remain constant at 65.53 GiB, demonstrating that memory savings come primarily from reduced KV cache rather than weight compression.

The hybrid architecture also benefits quantization in ways that pure attention models cannot. Recurrent layers require fewer bits to maintain quality because their state averaging provides inherent noise filtering. While attention layers with softmax operations are sensitive to precision loss (requiring FP8 or higher), recurrent layers can operate at INT4 or even INT2 without significant perplexity degradation. This differential sensitivity enables mixed quantization strategies where attention layers use FP8 and recurrent layers use lower precision, achieving additional memory savings beyond what uniform quantization provides.

6.4 Deployment Implications

For consumer GPUs (12GB–24GB), hybrid models like Qwen3.5-35B-A3B at 4-bit quantization (19.76GB weights) enable 48K–64K context, sufficient for coding and document analysis. On the DGX Spark (128GB), the same model supports 417K tokens with approximately 8× concurrency.

The optimal quantization strategy for hybrid architectures applies FP8 to attention layers (maintaining numerical stability for softmax operations) and INT4 or INT2 to recurrent layers (maximizing compression where state averaging provides inherent noise robustness). Our benchmarks confirm this approach: the hybrid GPTQ-INT4+FP8 configuration achieves 49% throughput improvement (22.4 tok/s) by converting 1085 BF16 tensors to FP8, reducing the 122B checkpoint to 74.19GB with 4.66GB savings.

Based on our results, we recommend three production configurations. First, consumer hardware (12GB–24GB) should target 48K–64K context with FP8/INT4 quantization, achieving 15–20 tok/s. Second, workstation hardware (48GB–80GB) enables 100K–200K context with 4–8× concurrency. Third, unified memory systems (128GB+) support 400K+ context with 8× or greater concurrency.

7 Discussion

7.1 Key Findings

Our experiments demonstrate that layer-aware quantization strategies yield measurable improvements over uniform approaches for specific deployment scenarios. The 49% throughput improvement from hybrid GPTQ-INT4+FP8 quantization on Qwen3.5-122B-A10B is primarily attributable to avoiding FlashInfer NVFP4 MOE crashes on SM121 architecture, rather than inherent quantization advantages. This highlights a practical reality: quantization strategy selection is often driven by software compatibility as much as by theoretical optimality.

The TQ3_0 KV cache compression effectively enables 200K+ context on 12GB GPUs, but the 24% throughput penalty (16.0 to 12.1 tok/s) means practitioners must weigh memory availability against latency requirements. For interactive applications where response time is critical, the q8_0 baseline at 14–15 tok/s with lower memory may be preferable to TQ3_0's maximum context capability.

The 48K context sweet spot for coding tasks is inherently workload-dependent. While our benchmarks on Qwen3.5-35B-A3B suggest 48K as optimal for code-related prompts, this finding may not generalize to other models, GPU configurations, or task types. Document analysis workloads, for instance, likely benefit from longer contexts despite throughput penalties.

7.2 Practical Deployment Guidelines

Based on our empirical results, we offer the following deployment recommendations:

Consumer GPUs (12GB–16GB): Use Q4_K_M weight quantization with q8_0 KV cache for interactive workloads. Enable TQ3_0 only when context exceeds 32K tokens. Configure 48K context as the default for coding tasks.

Edge Servers (128GB unified memory): Use hybrid GPTQ-INT4+FP8 for MoE models. This avoids FlashInfer NVFP4 crashes while providing 49% throughput improvement. Configure 85% GPU memory utilization for production.

Inference Framework Selection: llama.cpp is optimal for consumer GPU deployment with TQ3_0 support and offline use. vLLM provides superior multi-user serving through PagedAttention but requires more memory overhead.

7.3 Cost Analysis

An RTX 4080 Laptop GPU ($2,000–2,500) enables 9B–35B models at 12–16 tok/s, sufficient for individual developer use. A DGX Spark ($3,000–4,000) supports 122B models at 22.4 tok/s with 128GB memory, suitable for small team deployments. At typical cloud pricing ($2–4/hour for A100-equivalent), on-premise deployment becomes cost-effective within 12–18 months at 100 hours/month usage.

7.4 Limitations

Several limitations constrain the generalizability of our findings. First, our evaluation is restricted to the Qwen3.5 model family; results may differ for Llama, Mistral, or other architectures. Second, we report throughput and memory metrics but do not include perplexity measurements, making quality claims dependent on prior work. Third, the 48K context sweet spot is derived from a single model on a single GPU and should not be treated as a universal recommendation. Fourth, TQ3_0's implementation requires the vec-only kernel path with head dimension multiples of 64, limiting compatibility. Fifth, the hybrid quantization strategy assumes manual layer-type specification; automated layer classification remains future work.

7.5 Future Work

Priority directions include: perplexity benchmarks across all configurations, cross-platform evaluation (AMD, Intel, Apple Silicon), automated layer-type classification for hybrid quantization, integration of TurboAngle's Patel [2026] angle-domain approach for comparison with TQ3_0, and standardized benchmarks enabling reproducible comparison across quantization methods.

8 Conclusion

This empirical study evaluates layer-aware quantization strategies for deploying large language models on consumer and edge hardware. Key findings include: (1) TQ3_0, a llama.cpp implementation based on Google's TurboQuant, achieves 4.9× KV cache reduction enabling 200K context on 12GB GPUs, with a 24% throughput trade-off; (2) hybrid GPTQ-INT4+FP8 quantization delivers 49% throughput improvement on Qwen3.5-122B-A10B, largely by avoiding software compatibility issues with uniform NVFP4; (3) 48K tokens represents a practical context sweet spot for coding tasks on 12GB GPUs; and (4) hybrid memory architectures enable 417K context on 128GB unified memory through recurrent layer compression.

These results demonstrate that practical deployment decisions depend on workload characteristics, hardware constraints, and software compatibility—not solely on theoretical compression ratios. The quantization landscape evolves rapidly, with methods like TurboAngle Patel [2026] and TurboQuant Zandieh et al. [2025] advancing beyond the approaches evaluated here. Practitioners should treat our findings as deployment guidelines for the specific hardware and models tested, rather than universal recommendations.

Layer-aware hybrid quantization enables previously infeasible deployment scenarios from 122B models on desktops to 200K contexts on laptop GPUs, maintaining AI accessibility as models scale.

References

Yukai Chen, Ziyao Qiu, Shiyu Zhang, Jing Yang, et al. LongLoRA: Efficient fine-tuning of long-context large language models. *arXiv preprint arXiv:2309.12307*, 2023.

Marlin Contributors. Marlin: Fast 4-bit matrix multiplication for transformers, 2024. Optimized INT4 kernel library for NVIDIA GPUs. Available at: github.com/IST-DASLab/marlin.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized large language models. *arXiv preprint arXiv:2305.14314*, 2024.

Yiran Ding, Lyuhao Zhang, Chen Li, et al. LongRoPE: Extending LLM context window beyond 2 million tokens. *arXiv preprint arXiv:2402.13753*, 2024.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23:1–40, 2022.

B. J. Fino and V. R. Algazi. Unified matrix treatment of the fast Walsh-Hadamard transform. *IEEE Transactions on Computers*, C-25(11):1142–1146, 1976.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Georgi Gerganov and contributors. llama.cpp: LLM inference in C/C++. https://github.com/ggerganov/llama.cpp, 2023.

Robert M. Gray. Vector quantization. *IEEE ASSP Magazine*, 1(2):4–29, 1984.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Coleman Hooper, Sehoon Kim, Hiva Mohammad, Mahmoud Elhoushi, Hailey Huang, Ali Shafiee, Kurt Keutzer, and Amir Gholami. KVQuant: Towards 10 million context length LLM inference with KV cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.

Kathy J. Horadam. *Hadamard Matrices and Their Applications*. Princeton University Press, 2012.

Albert Q. Jiang, Alexandre Sablayrolles, Alexandre Roux, Arthur Mensch, Blanche Savary, Chris Bamler, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *Proceedings of the 37th International Conference on Machine Learning*, 2020.

Sehoon Kim, Amir Gholami, Zhewei Yao, Zhen Dong, Michael W. Mahoney, and Kurt Keutzer. Full stack optimization of transformer inference. *arXiv preprint arXiv:2308.00330*, 2023.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. vLLM: Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Opher Lieber, Barak Lenz, Andre Batazzi, Kosti Sberbank, Almog Aboud, et al. Jamba: A hybrid transformer-mamba language model. *arXiv preprint arXiv:2403.19887*, 2024.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware weight quantization for LLM compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2024.

Zirui Liu, Chao Zhao, Shiyi Tan, Zhiyu Fei, Yunan Zhou, Tianyi Li, Beibei Zhang, Chen Huang, and Zhi Ge. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. *arXiv preprint arXiv:2402.02750*, 2024.

Paulius Micikevicius, Dusan Stosic, Michael Harris, Pradeep Khadilkar, James McClure, et al. FP8 formats for deep learning. OCP (Open Compute Project) Specification, 2022.

Dipkumar Patel. TurboAngle: Near-lossless KV cache compression via uniform angle quantization. *arXiv preprint arXiv:2603.27467*, 2026.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Alber, Aamod Bhatt, Ankur Bhatt, et al. RWKV: Reinventing RNNs for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023a.

Houwen Peng, Kan Wu, Yixuan Wei, Guosheng Chen, Bin Gao, Kaipeng Wong, Shuming Zhang, et al. FP8-LM: Training FP8 large language models. *arXiv preprint arXiv:2310.18313*, 2023b.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kate Xiao, et al. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems (MLSys)*, 5, 2023.

Liliang Ren, Hang Liu, Yutong Wu, Shuwen Xie, Lu Yu, and Juntao Li. Samba: Simple hybrid state space models for efficient unlimited context language modeling. *arXiv preprint arXiv:2406.07522*, 2024.

Noam Shazeer. Fast transformer decoding: One write-head is more than enough. *arXiv preprint arXiv:1911.02150*, 2019.

Ying Sheng, Lianmin Zheng, Bin Li, Joseph McClelland, Mihai Christodorescu, et al. FlexGen: High-throughput generative inference of large language models with a single GPU. *arXiv preprint arXiv:2303.06865*, 2023.

Mingjie Sun, Zhiyu Li, Xin Cui, and Shuming Wang. Training and inference with large language models using FP8. *arXiv preprint arXiv:2401.13979*, 2024.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, 2024.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

Amir Zandieh, Majid Daliri, Majid Hadian, and Vahab Mirrokni. TurboQuant: Online vector quantization with near-optimal distortion rate. *arXiv preprint arXiv:2504.19874*, 2025.

Neil Zeghidour, Alejandro Luebs, Ahmed Omran, Jan Skoglund, and Marco Tagliasacchi. SoundStream: An end-to-end neural audio codec. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 30:495–507, 2021.