KVQuant: Adaptive Long-Context KV-Cache Compression for Memory-Bounded LLM Inference

Aman Sachan

Abstract

Large language models spend substantial memory on the key-value (KV) cache during autoregressive generation. The cache grows linearly with sequence length, batch size, number of layers, and attention heads, frequently becoming the dominant memory bottleneck at long context lengths. We present KVQuant, a cache-compression framework that reduces KV memory via adaptive mixed-precision quantization with lightweight dequantisation and optional forward-pass wrapping. KVQuant assigns higher precision to recent tokens and lower precision to older tokens, reflecting the empirical asymmetry of token importance in causal generation. We describe the design, provide memory and error models, and evaluate the implementation on synthetic cache tensors and wrapped-model integration tests. On representative 4-bit settings, the system achieves approximately 4× theoretical KV-memory reduction relative to fp16 storage while preserving stable round-trip reconstruction in unit tests. The project is positioned as a practical inference-time system for long-context LLMs rather than a training-time compression method.

1 Introduction

The KV cache is one of the most underestimated costs in LLM inference. Model weights get most of the attention, but for long prompts and multi-turn dialogue, the cache can quietly become the thing that breaks deployment. Let a transformer have batch size B, layers L, heads H, head dimension D, sequence length T, and bytes per element b. Then a standard fp16 cache scales as

$$M_{KV} = 2 \cdot B \cdot L \cdot H \cdot T \cdot D \cdot b.$$

The factor 2 appears because both key and value tensors are stored. This linear growth is brutal for long-context systems.

KVQuant attacks that bottleneck directly. Instead of treating all cache positions as equally important, it uses adaptive precision: recent tokens get more bits, older tokens get fewer. The intuition is simple: causal attention is not uniformly flat over time, and a one-size-fits-all bit budget wastes memory.

2 Related Work

KV-cache compression has become a serious research area because long-context inference is expensive. Representative directions include asymmetric low-bit quantization for cache tensors, heavy-hitter / token eviction approaches, adaptive precision allocation by position or importance, and hybrid schemes that preserve sensitive tokens at higher fidelity.

KVQuant is closest in spirit to adaptive cache quantization, but the implementation is intentionally practical: it is a drop-in Python package with explicit compression metadata, dequantisation routines, cache wrapping, and benchmarkable statistics.

3 System Overview

KVQuant has three layers:

1. bit allocation over sequence positions,
2. tensor quantization into compact integer representations,
3. cache wrapping so compression can happen automatically in a model loop.

The public API exposes `AdaptiveQuantizer`, `Dequantizer`, `KVQuant`, and `CompressedKVLayer`.

4 Bit Allocation Design

KVQuant currently uses a simple monotone schedule. For a cache of length T, define bit widths:

$$b_r \text{ (recent window)}, \quad b_m \text{ (middle window)}, \quad b_o \text{ (old window)},$$

with $b_r \geq b_m \geq b_o$.

Conceptually,

$$\beta_t = \begin{cases} b_r & \text{if } t \in \text{recent window} \\ b_m & \text{if } t \in \text{middle window} \\ b_o & \text{otherwise.} \end{cases}$$

This is a coarse but defensible approximation to importance-aware compression. It matches the common observation that recent context is usually more fragile than older context, while still leaving room for a practical fallback.

5 Quantization Model

For a tensor $X \in \mathbb{R}^n$, uniform affine quantization maps it to integers:

$$Q(X) = \text{clip}\left(\left\lfloor \frac{X - x_{\min}}{s} \right\rceil, 0, 2^b - 1\right),$$

where

$$s = \frac{x_{\max} - x_{\min}}{2^b - 1}.$$

The reconstruction is

$$\hat{X} = Q(X) \cdot s + x_{\min}.$$

KVQuant stores the metadata needed to reverse this mapping: scale, zero point / minimum value, original shape, original dtype, original number of elements, and packed byte estimate.

5.1 Blockwise option

For tensors where local variation dominates, KVQuant also supports blockwise quantization. If the tensor is flattened into blocks of size B, then each block gets its own min/max statistics. This reduces quantization error at the cost of more metadata.

6 Cache Compression Pipeline

For each layer cache (K, V):

1. compute a bit allocation from sequence length,
2. quantize keys and values using that allocation,
3. store compressed tensors plus metadata,
4. on reuse, dequantize them back to floating point.

The framework uses a `CompressedKVLayer` wrapper so compressed tuples remain structured and easy to detect during later passes.

Algorithm 1: Adaptive KV cache compression.

Input: key states K, value states V, layer index ℓ
Output: compressed $\hat{K}$, compressed $\hat{V}$, metadata M
1. $T \leftarrow$ sequence length of K
2. $\beta \leftarrow$ allocate_bits(T)
3. $(\hat{K}, M_K) \leftarrow$ quantize_adaptive(K, β)
4. $(\hat{V}, M_V) \leftarrow$ quantize_adaptive(V, β)
5. $M \leftarrow \{\ell, \beta, M_K, M_V\}$
6. return $\hat{K}, \hat{V}, M$

Algorithm 2: Decompression.

Input: compressed tensors $\hat{K}, \hat{V}$, metadata M
Output: restored K, V
1. $K \leftarrow$ dequantize_adaptive$(\hat{K}, M_{key})$
2. $V \leftarrow$ dequantize_adaptive$(\hat{V}, M_{value})$
3. return K, V

7 Memory Analysis

If the original cache is fp16, each element costs 2 bytes. For 4-bit quantization, the theoretical per-element cost is 0.5 bytes, ignoring metadata. Thus the ideal ratio is

$$R = \frac{16}{b}.$$

So at 4 bits,

$$R = 4 \times .$$

That matches the benchmark output from the repository's synthetic memory estimator for several model configurations.

7.1 Example

For a representative configuration, the code estimates:

- Llama-3-8B: 4.00 GB original $\rightarrow$ 1.00 GB compressed,
- Llama-3-70B: 20.00 GB original $\rightarrow$ 5.00 GB compressed,
- Mistral-7B: 16.00 GB original $\rightarrow$ 4.00 GB compressed.

These are not claimed as full end-to-end production measurements; they are memory model estimates produced by the repository's benchmark script.

8 Error and Stability Considerations

Compression error depends on bit width and tensor distribution. For a quantized tensor, the reconstruction error is typically measured with mean absolute error or relative error:

$$\mathrm{MAE}(X, \hat{X}) = \frac{1}{n} \sum_i |x_i - \hat{x}_i|, \quad \mathrm{RelErr}(X, \hat{X}) = \frac{\mathrm{MAE}(X, \hat{X})}{\frac{1}{n} \sum_i |x_i|}.$$

The repository's benchmark script reports that, on synthetic tensors, 2-bit compression has large error, 3-bit improves substantially, 4-bit gives a reasonable balance, and 8-bit is near-lossless.

9 Implementation Notes

KVQuant is notable because it is not just a whiteboard idea. The codebase contains a low-level quantizer, a dequantizer, top-level compression orchestration, cache wrapping, tests for accuracy, memory reduction, and integration, plus benchmark tooling.

The repository currently passes its test suite, which is a good sign that the package is internally coherent.

10 Experimental Evidence

The current repository-level evaluation is synthetic rather than full-model deployment on a large benchmark set. That is fine for a technical report, as long as the scope is stated honestly.

Reported evidence includes quantization/dequantization round-trip tests, cache wrapping tests, memory reduction checks, benchmarked latency overhead on synthetic tensors, and theoretical memory savings across model configurations.

A proper paper version should add a real evaluation matrix:

- perplexity on long-context language modelling,
- throughput on a target hardware stack,
- memory footprint against fp16 and alternative baselines,
- quality-vs-compression trade-off curves.

11 Limitations

KVQuant is useful, but not magical.

- It is inference-oriented, not training-oriented.
- It assumes cache objects compatible with `past_key_values`-style flows.
- Its default adaptive rule is heuristic, not learned.
- It does not yet include custom CUDA kernels.
- It should be validated carefully on production models before being trusted.

12 Broader Significance

This project matters because it targets the hidden tax of long-context generation. Models are getting bigger, prompts are getting longer, and users expect chats to feel persistent. Cache compression is one of the few ways to make that experience affordable without changing the model itself.

The broader claim is not that KVQuant solves all memory pressure. The claim is more interesting: it demonstrates that a clean software-layer design can make long-context inference more tractable right now.

13 Conclusion

KVQuant provides a practical, auditable cache-compression pipeline for long-context LLM inference. Its main ideas are simple but effective: allocate bits by recency, quantize the cache, preserve enough metadata to reconstruct it, and integrate the whole thing into a wrapped forward pass. In the current codebase, this yields a credible $4\times$ theoretical memory reduction at 4-bit precision, backed by passing tests and benchmark scripts.

References

[1] KVQuant repository codebase.

[2] C. Lou et al. KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache. arXiv:2402.02750.

[3] Z. Zhang et al. H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. arXiv:2306.14048.

[4] Y. Xiao et al. Efficient Streaming Language Models with Attention Sinks. arXiv:2309.17453.