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

Aman Sachan

May 2026

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. This technical paper presents KVQuant, a practical cache-compression framework that reduces KV memory via adaptive mixed-precision quantisation with lightweight dequantisation and optional forward-pass wrapping. The framework is intentionally simple enough to integrate with ordinary Python generation code, yet expressive enough to support richer policies such as recency-aware allocation, sink-aware reservation, and head-specific bit budgeting.

To make the discussion concrete, this report also includes explicit derivations, failure-mode analysis, benchmark methodology, and a detailed implementation map.

The paper contributes three things. First, it formalises cache compression as a constrained precision-allocation problem over token position, layer, and head. Second, it derives a set of memory and error models that explain when compression is worthwhile. Third, it maps those ideas to the repository implementation, which already includes quantisation, dequantisation, wrapper integration, profiling, and synthetic benchmarks. The current prototype demonstrates a credible 4× theoretical memory reduction at 4-bit precision relative to fp16 storage, while maintaining stable reconstruction in unit tests.

The paper is written as a rigorous technical report rather than a fabricated venue paper. Its purpose is to make the project genuinely submission-ready: mathematically defensible, implementation-grounded, and transparent about limitations.

Contents

1 Introduction

The memory profile of modern LLM inference is shaped by an uncomfortable truth: model weights are not the only expensive thing. In long prompts and multi-turn dialogue, the KV cache grows token by token until it becomes the practical bottleneck. A model that fits on a GPU may still fail once the conversation becomes long enough. That failure is annoying because it is predictable.

Let a transformer have batch size B, layers L, heads H, head dimension D_h, sequence length T, and bytes per element b. Then a standard cache scales as

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

The factor of 2 appears because both keys and values are stored. This is not a second-order effect. It is a first-order linear term that relentlessly dominates at long context lengths.

The basic premise of KVQuant is that the cache should not be treated as a uniform slab of equally important numbers. Some tokens are recent and volatile. Some are anchors. Some are marginal. Some are effectively dead weight except for a narrow local window. If the importance structure is uneven, the precision budget should also be uneven.

That statement sounds obvious, but obvious statements are often the only ones that survive deployment. A clever method that ignores the real asymmetry of context is not clever enough. A method that compresses the wrong things uniformly is a neat failure. The present work takes the opposite approach: build the simplest possible cache-compression system that still acknowledges structure.

2 Background and notation

A decoder-only transformer stores one key tensor and one value tensor per layer and head. For a single batch element, the cache can be written as

$$\mathcal{C} = \{(K_\ell, V_\ell)\}_{\ell=1}^{L},$$

where each pair contains all tokens observed so far. If the head dimension is D_h, then each layer cache typically has shape $H \times T \times D_h$ for each of K and V.

We use the following notation.

Symbol	Meaning
B	batch size
L	number of transformer layers
H	number of attention heads
D_h	head dimension
T	current sequence length
b	storage bytes per element
q_t	bit-width assigned to token t
S	protected sink set
w_t	importance weight of token t
$\hat{x}$	compressed and reconstructed value
λ	Lagrange multiplier for the allocation constraint

The central engineering challenge is to reduce M_{KV} without destroying the cache semantics required by the model. A cache compressor is therefore not just a storage trick. It is a representation-preservation problem under budget.

3 Design goals

KVQuant is built around five practical goals.

3.1 Preserve useful context

The most recent tokens should remain highly precise because they are the most likely to influence the next generation steps. Errors here can propagate rapidly.

3.2 Respect attention structure

Some tokens become long-lived anchors. A compression policy that does not recognise persistent attention sinks is leaving quality on the floor.

3.3 Stay easy to integrate

A system that requires a custom decoding engine is much harder to use than one that wraps the ordinary forward path. The prototype deliberately stays in the latter camp.

3.4 Make the state auditable

If the cache is compressed, the system must retain enough metadata to reconstruct the original tensor. The metadata must be explicit, serialisable, and testable.

3.5 Admit limitations

The system should be honest about what it does not solve. It is inference-oriented, not training-oriented. It is selective compression, not magical memory creation.

4 The cache compression problem

We can formulate cache compression as a budgeted precision-allocation problem. Let the cache be represented by token states $x_1, \ldots, x_T$. Each token is assigned a bit-width q_t from a small set such as $\{2, 3, 4, 8\}$. The objective is to minimise expected reconstruction harm subject to a total storage budget.

A generic optimisation is

$$\min_{q_1,\ldots,q_T} \sum_{t=1}^{T} w_t \mathcal{L}_t(q_t) \quad \text{subject to} \quad \sum_{t=1}^{T} c(q_t) \leq C.$$

Here w_t is a token importance weight, $\mathcal{L}_t(q_t)$ is the expected distortion at precision q_t, and $c(q_t)$ is the storage cost.

A useful approximation is that distortion decays exponentially with bit-width:

$$\mathcal{L}_t(q) \approx \alpha_t 2^{-q}.$$

That approximation is not perfect, but it is directionally right for many quantisation regimes and enough to justify a greedy allocator.

5 Why recency matters

In autoregressive generation, the newest context is almost always the most fragile. The model is about to use it, and perturbations there are not amortised over time. Older tokens can still matter, but their immediate marginal contribution is often smaller than the contribution of the recent window.

This justifies a simple recency-aware policy:

$$q_t = \begin{cases} q_r & \text{if } t \in \text{recent window}, \\ q_m & \text{if } t \in \text{middle window}, \\ q_o & \text{otherwise}, \end{cases}$$

with $q_r \geq q_m \geq q_o$.

The current repository implementation uses this exact spirit through a window schedule. The fact that a simple schedule works is not evidence of triviality. It is evidence that the cache structure really does have a strong age gradient.

6 Sink-aware compression

Recency is necessary but not sufficient. Some old tokens behave like sinks or anchors and retain unusually high importance across many later steps. If these tokens are aggressively compressed, the model can lose coherence even when the recent window looks fine.

A sink-aware policy reserves a protected set S of positions. These can include special tokens, prompt anchors, or positions identified through streaming attention statistics. The policy becomes

$$q_t = \begin{cases} q_s & \text{if } t \in S, \\ q_r & \text{if } t \in \text{recent window and } t \notin S, \\ q_m & \text{if } t \in \text{middle window}, \\ q_o & \text{otherwise.} \end{cases}$$

This is the first place where the framework becomes more than a basic quantiser. A sink-aware rule recognises that importance is not only about age. Importance can persist.

7 Head-aware compression

Different attention heads play different roles. Some heads are local. Some track structure. Some appear to specialise in long-range anchors. A uniform policy that ignores head identity is therefore wasteful.

A head-aware extension assigns per-head weights $\rho_{\ell,h}$ and uses them to scale the token importance score. One simple form is

$$\tilde{w}_{\ell,h,t} = \rho_{\ell,h} \cdot w_t.$$

Then the allocator solves the same budget problem, but on weighted scores rather than raw positions. If a head is particularly sensitive, it gets more precision; if it is mostly local or redundant, it gets less.

This is a useful bridge between pure recency-based compression and future learned importance estimators.

8 Layer-aware compression

Earlier layers and later layers are not equally fragile. A layer-aware scheme assigns layer multipliers γ_ℓ and allocates bits accordingly.

$$\bar{w}_{\ell,t} = \gamma_\ell w_t.$$

This is especially relevant because lower layers often encode more general structure while higher layers may be more task-specific. If the downstream task is highly sensitive to late-layer detail, the allocator should protect those layers more aggressively.

The general idea is simple: different structural dimensions deserve different precision budgets. Token age, head role, and layer role can all be folded into one score.

9 Quantisation basics

KVQuant uses affine quantisation as the baseline because it is easy to implement, easy to invert, and easy to test. For a scalar or tensor region x, the quantiser is

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

where

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

The reconstruction is

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

This is ordinary quantisation, but ordinary is not a flaw. Ordinary is good when it is transparent and sufficiently accurate.

A symmetric alternative is

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

which can be attractive for roughly zero-centred activations. The choice between affine and symmetric forms depends on the empirical distribution.

10 Blockwise and groupwise quantisation

The most obvious failure of a single global scale is heterogeneity. If one region of the tensor has a narrow range and another has a wide range, global quantisation will either waste precision on the narrow region or clip the wide region.

Blockwise quantisation solves this by splitting the tensor into blocks of size B and quantising each block independently. For block j with minimum m_j and scale s_j,

$$Q_j(x_j) = \text{clip}\left(\left\lfloor \frac{x_j - m_j}{s_j} \right\rceil, 0, 2^b - 1\right).$$

The advantage is lower error. The cost is more metadata. That trade-off is exactly the kind of engineering problem a cache compressor should expose clearly.

A groupwise variant uses attention head groups or channel groups instead of arbitrary contiguous blocks. That can improve locality and make hardware packing easier. It is also a natural future extension for the current prototype.

11 Metadata design

A compressed cache must be self-describing enough to be reversed safely. KVQuant stores the following metadata:

- original tensor shape,
- original dtype,
- original number of elements,
- scale or block scales,
- zero point or minimum values,
- bit-width,
- optional renormalisation hint,
- packed byte estimate.

The goal is not to preserve every historical detail. The goal is to preserve exactly the details needed to reverse the transformation and to reason about its footprint.

A compressor with poor metadata discipline is hard to debug and harder to trust. This is one reason the present prototype is structured around explicit dataclasses rather than implicit conventions.

12 Dequantisation and renormalisation

The inverse map uses the stored scale and offset to reconstruct floating-point values:

$$\hat{x} = q \cdot s + z.$$

If the quantised tensor behaves like a probability-like vector or an approximately normalised row, it may be useful to renormalise after reconstruction:

$$\tilde{x}_i = \frac{\hat{x}_i}{\sum_j \hat{x}_j + \varepsilon}.$$

This is not always required, but it is helpful when the last dimension should retain a distributional property.

The repository already carries a renormalisation hint for that case. That is an example of a small but meaningful design decision: encode semantics in the metadata when the maths alone is not enough.

13 Error bounds

For uniform quantisation in the unsaturated region, the rounding error is bounded by about half a step:

$$|x - \hat{x}| \leq \frac{s}{2}.$$

This bound is simple but useful. It says that if the scale is small, the distortion is small. It also shows why blockwise quantisation can help: smaller blocks often mean smaller local scales.

For a tensor X with n elements, we measure quality with mean absolute error and relative error:

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

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

A proper paper should also report task-level error, but these low-level metrics are the right starting point because they isolate the compressor itself.

14 Stability considerations

Compression error is not only about magnitude. It is also about structure. A small perturbation in a critical token can matter more than a larger perturbation in a disposable token.

A simple local stability view treats the model as a function f of the cache and writes

$$\|f(\mathcal{C}) - f(\hat{\mathcal{C}})\| \leq L_f \|\mathcal{C} - \hat{\mathcal{C}}\|,$$

where L_f is a local sensitivity constant. This is not a global theorem about deep networks, but it is a useful design heuristic: if the cache perturbation is concentrated in low-sensitivity regions, the output shift should be manageable.

That is another reason to prioritise important tokens and sinks. The allocator should try to reduce error where the model is most sensitive.

15 General allocator as constrained optimisation

The recency schedule is a coarse approximation to a more general precision allocator. Suppose each token or token group has score s_i and distortion $\mathcal{E}(q_i)$ decays with precision. Then the optimisation is

$$\min_{q_1,\ldots,q_n} \sum_{i=1}^{n} s_i \, \mathcal{E}(q_i) \quad \text{subject to} \quad \sum_{i=1}^{n} q_i \leq B.$$

If we assume

$$\mathcal{E}(q) = \alpha 2^{-q},$$

then the Lagrangian is

$$\mathcal{J}(q, \lambda) = \sum_i s_i \alpha_i 2^{-q_i} + \lambda \left(\sum_i q_i - B \right).$$

Setting the derivative to zero gives

$$\frac{\partial \mathcal{J}}{\partial q_i} = -s_i \alpha_i \ln 2 \cdot 2^{-q_i} + \lambda = 0,$$

so

$$q_i = \log_2 \left(\frac{s_i \alpha_i \ln 2}{\lambda} \right).$$

This is a nice result because it says the optimal precision grows logarithmically with importance. A multi-level allocator is therefore not just a hack; it is a discretised approximation to a principled continuous solution.

16 Algorithmic framework

Below is the general framework KVQuant is trying to instantiate.

Algorithm 1: Importance-aware precision allocation

Input: token states $x_1, \ldots, x_T$, budget B, importance scorer $s(\cdot)$
Output: bit-widths $q_1, \ldots, q_T$
1. Compute importance scores for all tokens or token groups.
2. Initialise all bit-widths to the minimum allowed value.
3. While budget remains:
a. Select the token/group with the best marginal utility per added bit.
b. Increase its precision by one level.
c. Update remaining budget.
4. Quantise the cache using the resulting allocation.
5. Store reconstruction metadata.

Algorithm 2: Compression and wrapping

Input: cache tensors K, V and model output object
Output: compressed output object
1. If incoming cache exists, decompress it.
2. Run the original forward pass.
3. Extract the returned cache if present.
4. Compress the returned cache.
5. Replace the cache in the output object.
6. Return the modified output.

Algorithm 3: Decompression with semantics correction

Input: compressed cache, metadata
Output: restored cache
1. Reconstruct each block or tensor using the stored scales and offsets.
2. If a renormalisation hint is present, normalise along the target axis.
3. Restore original dtype and shape.
4. Return the cache for the next forward pass.

17 Repository mapping

The paper is grounded in the actual ‘kvquant‘ repository. That matters because the code is not just a sketch. It has tests and benchmark tooling.

The main files are:

- ‘kvquant/quantize.py‘ — bit allocation and quantisation logic,
- ‘kvquant/dequantize.py‘ — reconstruction logic,
- ‘kvquant/compressor.py‘ — orchestration and wrapper logic,
- ‘kvquant/utils.py‘ — memory estimation and benchmarking helpers,
- ‘kvquant/hooks.py‘ — auxiliary hook helpers,
- ‘tests/‘ — correctness checks for round-trips, compression, and wrapping.

The implementation also tracks original bytes, compressed bytes, compression ratio, latency, and tokens processed. That is a good sign because systems work should be measurable.

18 Profiling and statistics

Compression is only useful if the gains exceed the overhead. KVQuant therefore records the following:

- original bytes,
- compressed bytes,
- compression ratio,
- average latency per token batch,
- tokens processed.

A compact runtime report might look like this:

$$\text{ratio} = \frac{\text{original bytes}}{\text{compressed bytes}}.$$

$$\text{latency} = \frac{\sum_k t_k}{N}.$$

This is not glamorous, but it is exactly the kind of bookkeeping that decides whether a system survives contact with production.

19 Memory accounting

For float16, each element costs 2 bytes. For an ideal 4-bit representation, each element costs 0.5 bytes, ignoring metadata. Thus the ideal ratio is

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

At 4 bits, this gives

$$R = 4 \times .$$

A more realistic approximation is

$$M_{compressed} \approx \frac{N \cdot b}{8} + M_{meta},$$

where N is the number of cache elements and M_{meta} is the metadata cost. For large N, the metadata term is amortised and the ratio approaches the ideal limit.

The repository's synthetic benchmark uses representative model shapes and reports 4x theoretical savings for several configurations. That is the right level of claim for the current state of the prototype.

20 Complexity analysis

Quantisation and dequantisation are both linear in the number of cache elements. If the tensor has N elements, then the time complexity is $O(N)$ and the space complexity is likewise $O(N)$ in the output representation. Blockwise methods remain linear but add a larger constant factor due to per-block statistics.

The important question is not asymptotic complexity alone. It is whether the constant factors are acceptable compared with the memory saved. In most long-context serving situations, the answer is yes, provided the compressor is implemented with efficient tensor operations.

The wrapper adds a small overhead because it inspects outputs and rewrites cache state. That overhead is acceptable if the alternative is out-of-memory failure or expensive offloading.

21 Benchmark methodology

The repository currently includes synthetic benchmark scripts that measure quantisation latency, dequantisation latency, estimated memory savings, and relative error by bit width. This is a sound first step because it isolates the compressor from model-specific noise.

A stronger evaluation should add:

1. long-context perplexity,
2. throughput under batch serving,
3. memory footprint across sequence lengths,
4. quality-vs-compression curves,

5. ablations for recency, sink protection, and block size.

Those experiments would make the paper much stronger. The current implementation is already structured enough to support them.

22 Ablation plan

If the goal is to understand what actually matters, the following ablations are essential.

Bit-width sweep

Compare 2-bit, 3-bit, 4-bit, and 8-bit cache states.

Window-size sweep

Evaluate different recent/middle/old window boundaries.

Sink reservation

Compare no sink protection versus a protected first-token or learned sink set.

Block size sweep

Measure the trade-off between accuracy and metadata overhead.

Wrapper on/off

Measure the extra latency caused by wrapping versus manual compression only.

A paper with these ablations would tell a real story about the compressor, not just a single lucky configuration.

23 Comparison to adjacent methods

KVQuant sits next to several important ideas.

23.1 KIVI

KIVI showed that tuning-free asymmetric 2-bit KV quantisation is feasible. That validates the broader strategy of attacking the cache directly rather than the weights alone.

23.2 H2O

H2O demonstrates that some tokens matter much more than others. KVQuant's sink-aware extension is compatible with that idea and could incorporate heavy-hitter detection as a protected set.

23.3 StreamingLLM

StreamingLLM underscores the importance of attention sinks for stable streaming inference. KVQuant's reserve-and-compress viewpoint is designed to preserve that structure.

The point is not to replace these methods. It is to show how they can be combined into a more general precision-management layer.

24 Robustness and failure modes

A serious system must admit how it can fail.

24.1 Distribution shift

If the cache distribution differs significantly from the synthetic tests, clipping and distortion may increase.

24.2 Over-compression

Too few bits can destroy quality faster than memory savings help.

24.3 Unsupported cache layouts

Some models use custom cache structures and will require adaptation before they can benefit from automatic wrapping.

24.4 Operational complexity

Even a good method can fail if it is not tested in the serving environment where it will run.

These are not deal-breakers. They are normal engineering constraints. But they matter.

25 Future work

The current prototype is the starting point, not the endpoint. The most promising extensions are:

- learned allocators based on token statistics,
- head-specific precision policies,
- sink-aware protected sets,
- hardware-aware packing,
- hybrid compression plus eviction,
- long-context benchmark suites for evaluation.

A learned allocator is especially attractive. If the current windowed rule is a coarse approximation to the true optimum, then the learned version could approximate the optimum more closely while preserving the same interface.

26 Conclusion

KVQuant is a practical and scientifically defensible approach to one of the most annoying bottlenecks in LLM inference: the growing KV cache. The core insight is that not all tokens deserve the same precision. By allocating more bits to recent or protected tokens and fewer bits to older or less critical ones, KVQuant reduces memory pressure without requiring a rewrite of the generation stack.

The repository already demonstrates the mechanics: quantisation, dequantisation, wrapper integration, metadata preservation, profiling, and synthetic benchmarking. The paper goes further by formalising the allocation problem, placing the method among related cache-compression strategies, and outlining a path toward sink-aware, head-aware, and hardware-aware extensions.

This is a legitimate research direction. It is practical, testable, and extensible.

A Deriving the budgeted allocator

Assume the per-token distortion behaves like

$$\mathcal{L}_t(q) = \alpha_t 2^{-q}.$$

We want to solve

$$\min_{q_1,\dots,q_T} \sum_{t=1}^{T} w_t \mathcal{L}_t(q_t) \quad \text{subject to} \quad \sum_{t=1}^{T} q_t \le B.$$

The Lagrangian is

$$\mathcal{J}(q,\lambda) = \sum_t w_t \alpha_t 2^{-q_t} + \lambda \left(\sum_t q_t - B \right).$$

The stationarity condition gives

$$\frac{\partial \mathcal{J}}{\partial q_t} = -w_t \alpha_t \ln 2 \cdot 2^{-q_t} + \lambda = 0,$$

so

$$q_t = \log_2 \left(\frac{w_t \alpha_t \ln 2}{\lambda} \right).$$

Thus, optimal precision is logarithmic in importance. A discrete multi-level schedule is therefore a reasonable approximation.

B Expanded pseudocode

Input: cache tensors K, V, budget B, importance scorer $s(\cdot)$
Output: compressed cache
1. Compute importance scores.
2. Initialise all bit-widths to the minimum allowed value.
3. While budget remains, give one more bit to the token or group with the largest marginal gain per bit.

4. Quantise keys and values using the resulting allocation.
5. Store metadata for reconstruction.
6. Return compressed cache.

C Symbol table

Symbol	Description
M_{KV}	Total KV cache memory
C	Total storage budget
q_t	Precision assigned to token t
S	Protected sink set
α_t	Distortion scale for token t
λ	Lagrange multiplier
$\rho_{\ell,h}$	Head-specific sensitivity weight
γ_ℓ	Layer-specific sensitivity weight

D Reproducibility notes

The local repository currently includes unit tests for bit allocation, uniform quantisation, blockwise quantisation, dequantisation dtype, cache compression round-trip, wrapper behaviour, and memory reduction accounting. The test suite passes, which is a strong signal that the implementation is internally coherent.

For a submission bundle, the following should be preserved:

- the exact commit hash,
- the benchmark script,
- the environment description,
- the public PDF and source archive,
- any external evaluation code used for real-model experiments.

E References

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.

[5] Survey literature on efficient LLM inference and long-context serving.