# TurboQuant Explained: How Google's KV Cache Trick Cuts Memory 6x With Zero Quality Loss

March 30, 2026 · by Mark Bartlett

Download this guide as PDF

**Quick Answer:** TurboQuant is a KV cache compression algorithm from Google that shrinks the memory your model uses for context by 4-6x with no accuracy loss. It doesn't touch model weights -- just the cache that grows as your conversation gets longer. llama.cpp and MLX implementations exist but aren't merged yet. Once they land in Ollama and LM Studio (estimated Q3 2026), you'll be able to run longer contexts on the same GPU without buying new hardware.

📚 **More on this topic:** VRAM Requirements Guide · What Can You Run on 24GB? · Context Length Explained · llama.cpp vs Ollama vs vLLM

Every time you send a message to a local LLM, the model stores information about every token it has read so far. That storage is the KV cache, and on a 24GB GPU running Qwen 3.5 27B at 32K context, it can eat 4-6GB of your VRAM – memory that could otherwise hold a larger model or a longer conversation.

Google published a paper called TurboQuant that compresses that cache down to 3-4 bits per element instead of the usual 16. The result: 4-6x less memory for context, 8x faster cache lookups, and zero accuracy loss on benchmarks. The paper dropped on arXiv in April 2025, Google blogged about it in March 2026, and it’s scheduled for presentation at ICLR 2026.

For anyone running local models on consumer GPUs, this is the most useful inference optimization to come along in months. Here’s how it works, what it means for your hardware, and when you’ll actually be able to use it.

---

## What the KV cache is (and why it eats your VRAM)

When a model processes your prompt, it doesn’t just read and forget. For every token it sees, it creates two vectors: a **key** (a label for that token’s role in context) and a **value** (the actual meaning data). These get stored in a lookup table – the KV cache – so the model can refer back to earlier parts of the conversation without reprocessing everything.

The problem: this cache grows linearly with context length. More tokens in the conversation means more key-value pairs stored. At FP16 precision, an 8B model at 32K context uses about 4.6GB just for the cache. A 70B model at 100K context can burn through 20GB+ of cache alone.

Here's what that looks like on real hardware:

| Model | Context length | KV cache (FP16) | KV cache (TurboQuant 4-bit) |
|---|---|---|---|
| Llama 3.1 8B | 4K | 640 MB | ~120 MB |
| Llama 3.1 8B | 32K | 4.6 GB | ~870 MB |
| Llama 3.1 8B | 128K | 18.4 GB | ~3.5 GB |
| Qwen 3.5 27B | 32K | ~5.4 GB | ~1.0 GB |

That 128K row is the one that matters. Without TurboQuant, running Llama 3.1 8B at its full 128K context on a 24GB GPU means the cache alone takes 18.4GB, leaving barely 6GB for model weights. With TurboQuant, the cache drops to 3.5GB, leaving 20GB for the model. That's the difference between "impossible" and "comfortable."

---

## How TurboQuant works

TurboQuant has two parts, and the analogy that helped me understand it is giving directions.

### PolarQuant: better coordinates

Standard quantization takes a vector of numbers and rounds each one to fit in fewer bits. Think of it like giving someone turn-by-turn directions: "go 3 blocks north, 2 blocks east, 1 block south." Each step gets rounded, and the errors accumulate.

PolarQuant converts the vectors from Cartesian coordinates to polar coordinates first. Instead of step-by-step directions, you point at the destination and say "that direction, this far." A direction (angle) and a distance (magnitude). After a random preconditioning step, these polar values follow a predictable statistical distribution, which means you can quantize them more efficiently with less distortion.

The practical win: PolarQuant eliminates the per-block normalization constants that traditional quantization methods need to store. No calibration data required. No fine-tuning. You just apply it at inference time and it works.

On its own, PolarQuant achieves 4.2x compression with quality that matches or beats existing methods like KIVI and KVQuant.

### QJL: 1-bit error correction

The second piece is Quantized Johnson-Lindenstrauss, or QJL. It applies a mathematical transform and then reduces each value to a single bit (+1 or -1). This creates an unbiased estimate of the residual error left by PolarQuant.

Together, PolarQuant + QJL achieve 6x compression at 3.5 bits per element with zero accuracy loss on LongBench, needle-in-haystack, and other standard benchmarks.

One interesting finding from the community: most implementers have dropped QJL entirely and put all available bits into better PolarQuant centroids instead. In practice, allocating bits to higher-quality centroids outperforms the theoretical elegance of the 1-bit error correction layer. The math is sound, but raw quality from better centroids wins.

---

## What this actually means for your GPU

Let’s be specific about what TurboQuant does and doesn’t change.

### What it does

**Longer context on the same hardware.** If you’re running Qwen 3.5 27B Q4_K_M on a 24GB GPU, the model weights take ~17GB. That leaves 7GB for the KV cache. At FP16, that’s roughly 40K tokens of context. With TurboQuant at 4-bit, you get 160K+ tokens in the same space.

**More room for larger models.** If a model barely fits in your VRAM with no context headroom, TurboQuant gives you breathing room. A model that previously choked at 8K context might run comfortably at 32K.

**Faster cache lookups on long conversations.** Google claims 8x speedup on attention logit computation with 4-bit cache vs FP32 on H100s. On consumer GPUs the improvement will be smaller, but the direction is real – less data to read means faster reads.

### What it doesn’t do

**Does not compress model weights.** Your Q4_K_M model still takes the same VRAM for weights. TurboQuant only compresses the cache that builds up during inference. If a model doesn’t fit in VRAM before loading any context, TurboQuant won’t help.

**Does not speed up training.** This is inference-only.

**Does not change the model itself.** Same weights, same architecture, same outputs. Just smarter storage for the intermediate state.

**The “8x speedup” is for cache operations, not overall tok/s.** Your total token generation speed depends on many things – memory bandwidth, compute, model size. The cache speedup is one component. Real-world tok/s improvement is meaningful but not 8x across the board.

---

## Community benchmarks

Google tested TurboQuant on H100s with Gemma, Mistral, and Llama. The community has since tested it on consumer hardware.

### Official results (Google, H100)

| Method | KV bits | LongBench score | Needle-in-haystack |
|---|---|---|---|
| Full precision (FP16) | 16 | 50.06 | 0.997 |
| TurboQuant | 3.5 | 50.06 | 0.997 |
| TurboQuant | 2.5 | 49.44 | 0.997 |

At 3.5 bits, the scores are identical. At 2.5 bits, there’s a slight LongBench dip but needle-in-haystack accuracy holds.

### Community results (consumer hardware)

**RTX 4080, Qwen2.5 models (back2matching/turboquant):**

| Model | Context | TQ 4-bit speed | FP16 speed | VRAM saved |
|---|---|---|---|---|
| Qwen2.5-3B | 4K | 7.4 tok/s | 2.5 tok/s | 1,048 MB |
| Qwen2.5-7B | 1.8K | 1.4 tok/s | OOM | 444 MB |
| Qwen2.5-0.5B | 8K | 19.8 tok/s | – | 2,070 MB |

That Qwen2.5-7B row is telling. At FP16 it ran out of memory entirely. With TurboQuant 4-bit it runs, period. Not fast, but it runs.

**Apple Silicon, Llama 3 8B (helgklaizar/turboquant_mlx):**

| Context | Uncompressed KV | TQ 3-bit | Compression |
|---|---|---|---|
| 4K | 64 MB | 12 MB | 5.3x |
| 64K | 1,024 MB | 192 MB | 5.3x |
| 128K | 2,048 MB | 384 MB | 5.3x |

Consistent 5.3x compression across all context lengths. On a MacBook with 36GB unified memory, that's the difference between running Llama 3 8B at 64K context uncomfortably and running it at 128K context with room to spare.

### Quality at different bit-widths

The community consensus after months of testing:

| Bits | Compression | Quality impact |
|---|---|---|
| 4-bit | 3.8x | Indistinguishable from FP16 on 3B+ models |
| 3.5-bit | 4.9x | Zero loss on 8B+, minor on smaller models |
| 3-bit | 4.9x | Noticeable on models under 8B |
| 2-bit | 7.1x | Visible degradation, fine for drafts |
| 1-bit | 12.8x | Research curiosity, not practical |

The sweet spot is 4-bit for reliability or 3.5-bit if you're willing to accept tiny quality variance on smaller models.

---

## Implementation status (where can you use it today?)

### llama.cpp

Not merged into mainline. There's an active discussion (#20969) and a feature request (#20977) with 212 upvotes, but no maintainer commitment on a timeline.

Several community forks exist:

- **TheTom/llama-cpp-turboquant** – the most mature fork, with Flash Attention support and `turbo3` / `turbo4` KV cache types on Metal
- **spiritbuun's fork** – CUDA support for RTX 3090+

- **Madreag's fork** – RTX 5090 tested, 4.6x KV compression, ~98% of q8_0 prefill speed

If you build llama.cpp from source and don't mind running a fork, you can use TurboQuant today. If you wait for the official merge, estimated timeline is Q2-Q3 2026.

### MLX (Apple Silicon)

Multiple working implementations:

- **helgklaizar/turboquant_mlx** – the most production-ready, with an OpenAI-compatible API, tested on DeepSeek R1 Distill 8B, Mistral Nemo 12B, Llama 3/3.2
- A HuggingFace model exists for Qwen3.5-35B with TurboQuant KV compression, showing exact-match quality from 8.5K to 64K context

If you're on Apple Silicon and comfortable with MLX, this is usable now.

### Ollama and LM Studio

Neither supports TurboQuant yet. Ollama has a feature request (#15051) with 117 upvotes and no maintainer response. Both tools pull from llama.cpp, so TurboQuant support will follow once the llama.cpp merge happens.

**Current workaround:** Use Q4_K_M quantization with `OLLAMA_FLASH_ATTENTION=1` to reduce VRAM usage from the compute side. Not the same as TurboQuant, but it helps.

### Python (pip install)

The fastest way to try TurboQuant today:

```
pip install turboquant
```

The back2matching/turboquant package drops into any HuggingFace model. Version 0.3.0 supports asymmetric K/V compression and layer-adaptive precision (keeps sensitive early/late layers at FP16). Works on any GPU with PyTorch.

---

## The Jevons Paradox angle

Memory chip stocks dropped 3-6% when Google blogged about TurboQuant. The logic: if you need 6x less cache memory, you need fewer GPUs. That logic is wrong.

Cheaper inference doesn't mean less GPU demand. It means people run longer contexts, load bigger models, and find new use cases that weren't worth the memory cost before. An RTX 3090 that could handle 32K context will now handle 128K – and users will immediately start using 128K. A 24GB card that forced you into Q4 for a 27B model now lets you use Q6 with the memory savings from cache compression.

This pattern – efficiency gains increasing total demand – plays out every time compute gets cheaper. For local AI users, TurboQuant is pure upside: same hardware, more capability. For the industry as a whole, it accelerates adoption.

---

## What to do now

If you build from source and want to experiment, grab one of the llama.cpp forks or `pip install turboquant` and test it on your models. The community implementations are stable enough for testing, not production.

If you use Ollama or LM Studio, wait. The llama.cpp merge will come, and your tools will pick it up automatically. This is weeks to months, not years.

The practical takeaway: TurboQuant doesn't let you run models that don't fit on your GPU. What it does is make the models that already fit run with massively longer context windows and slightly more headroom. If you've been hitting the wall at 32K context on a 24GB card, this is the fix that's coming – and it won't cost you a dime in new hardware.

---

## Related guides

- How Much VRAM Do You Need for Local LLMs?
- What Can You Run on 24GB VRAM?
- Context Length and Memory Explained
- llama.cpp vs Ollama vs vLLM
- Best Local Coding Models 2026

- LLM Quantization Explained

Get notified when we publish new guides.

Subscribe — free, no spam

---

Source: https://insiderllm.com/guides/turboquant-kv-cache-compression-local-ai/