KV Cache: Why Context Length Eats Your VRAM (And How to Fix It)

February 23, 2026 · by Mark Bartlett

Download this guide as PDF

Quick Answer: The KV cache stores attention state for every token in your context window, and it grows linearly with context length. For Llama 3.1 8B at FP16, the KV cache alone eats ~4GB at 32K context and ~16GB at 128K — more than the model weights at Q4. This is why your model loads fine but OOMs when you start a long conversation. The fastest fixes: reduce context length to what you actually need, enable Q8 KV cache quantization in llama.cpp (halves KV size with minimal quality loss), or use a model with Grouped Query Attention which already cuts KV cache 4-8x compared to older architectures.

📚 **More on this topic:** Context Length Explained · VRAM Requirements Guide · Context Length Exceeded Fix · Planning Tool

You loaded Llama 3.1 8B at Q4. It fits in 5GB. You've got 24GB of VRAM. Life is good. Then you set context to 128K because the model card says it supports it, and your system grinds to a halt or OOMs outright.

The model didn't get bigger. The KV cache did.

The KV cache is the most common source of "why doesn't this fit?" confusion in local AI. It's invisible in model specs, it doesn't show up in the GGUF file size, and most frontends don't tell you how much VRAM it's eating. At long context lengths, the KV cache can consume more memory than the model weights themselves.

This guide gives you the exact formula, worked examples for popular models, and six practical techniques to cut it down to size.

What the KV cache actually is

Every transformer model uses an attention mechanism to figure out how each token relates to every other token. During generation, the model computes Key and Value vectors for each token in the sequence. These vectors need to be stored so the model can reference them when generating the next token.

That stored set of Key and Value vectors is the KV cache.

Without it, the model would have to recompute attention across the entire context for every single new token. The KV cache is a speed optimization: trade memory for not recomputing the same thing thousands of times. That memory cost adds up fast.

Why it grows linearly

Every new token in the context adds one more set of Key and Value vectors to the cache. Double the context length, double the KV cache. This is fundamentally different from model weights, which stay the same regardless of how long your conversation is.

At short contexts (2K-4K tokens), the KV cache is negligible, maybe a few hundred megabytes. At long contexts (32K-128K tokens), it becomes the dominant VRAM consumer.

The KV cache formula

This is the exact formula. Our Planning Tool uses the same math under the hood:

```
KV cache (bytes) = batch_size × context_length × 2 × num_layers × num_kv_heads × head_dim × bytes
```

Breaking it down:

- **batch_size**: Number of parallel sequences. For local inference, this is almost always 1.
- **context_length**: Total tokens in the context window (your prompt + history + response).
- **2**: Two matrices (one for Keys, one for Values).
- **num_layers**: Number of transformer layers in the model.
- **num_kv_heads**: Number of Key-Value attention heads. This is NOT the same as total attention heads in GQA models.
- **head_dim**: Dimension of each attention head. Typically 128 for modern models.
- **bytes_per_param**: 2 for FP16, 1 for Q8 cache, 0.5 for Q4 cache.

Worked example: Llama 3.1 8B at 32K context

Llama 3.1 8B specs:

- 32 transformer layers
- 8 KV heads (uses GQA with 32 query heads but only 8 KV heads)
- 128 head dimension
- FP16 KV cache (default)

```
KV cache = 1 × 32,768 × 2 × 32 × 8 × 128 × 2
         = 1 × 32,768 × 2 × 32 × 8 × 128 × 2
         = 4,294,967,296 bytes
         = 4.0 GB
```

Four gigabytes. Just for the KV cache. The model weights at Q4_K_M are about 4.9GB. So at 32K context, you're nearly doubling your total VRAM requirement compared to minimum context.

Now scale that to 128K

```
KV cache at 128K = 1 × 131,072 × 2 × 32 × 8 × 128 × 2
                 = 17,179,869,184 bytes
                 = 16.0 GB
```

Sixteen gigabytes for the cache alone. Add 4.9GB for Q4 model weights plus ~0.5GB overhead, and you need **~21.5GB** just to load this 8B model at full context. That barely fits on an RTX 3090/4090, and this is an 8B model.

This is the answer to "why does my model support 128K context but I can't actually use it?"

Grouped query attention: why modern models aren't worse

If you look at the formula, the big variable is **num_kv_heads**. This is where Grouped Query Attention (GQA) saves you.

MHA vs. GQA vs. MQA

Older models used Multi-Head Attention (MHA), where every attention head gets its own Key and Value vectors. Newer models use Grouped Query Attention (GQA), where multiple query heads share the same KV heads.

Attention Type	KV Heads / Query Heads	KV Cache Multiplier	Used By
MHA	Equal (e.g., 32/32)	1x (baseline)	GPT-2, older models
GQA	Grouped (e.g., 8/32)	**0.25x**	Llama 3, Qwen 2.5, Mistral
MQA	Single (e.g., 1/32)	**0.03x**	Falcon, some StarCoder

Llama 3.1 8B uses 8 KV heads with 32 query heads, a 4x reduction in KV cache compared to MHA. If this model used MHA instead, that 32K KV cache would be **16GB instead of 4GB**. GQA is the reason modern models can advertise long context windows at all.

Llama 3.1 70B uses 8 KV heads with 64 query heads, an 8x reduction. Without GQA, 70B models would need absurd amounts of VRAM for any real context length.

KV cache sizes for popular models

All values are FP16 KV cache with batch size 1. These are the KV cache alone; add model weights and overhead separately.

Model	Layers	KV Heads	Head Dim	4K	8K	16K	32K	64K	128K
Llama 3.1 8B	32	8	128	0.5 GB	1.0 GB	2.0 GB	4.0 GB	8.0 GB	16.0 GB
Llama 3.1 70B	80	8	128	1.3 GB	2.5 GB	5.0 GB	10.0 GB	20.0 GB	40.0 GB
Qwen 2.5 14B	48	8	128	0.8 GB	1.5 GB	3.0 GB	6.0 GB	12.0 GB	24.0 GB
Qwen 2.5 32B	64	8	128	1.0 GB	2.0 GB	4.0 GB	8.0 GB	16.0 GB	32.0 GB
Mistral 7B	32	8	128	0.5 GB	1.0 GB	2.0 GB	4.0 GB	8.0 GB	16.0 GB
Mixtral 8x7B	32	8	128	0.5 GB	1.0 GB	2.0 GB	4.0 GB	8.0 GB	16.0 GB

A few things jump out:

- **Llama 3.1 70B at 128K** needs 40GB just for KV cache. At Q4, the weights are also ~40GB, so the KV cache matches the model size. You'd need ~80GB+ VRAM for this configuration.
- **Mixtral 8x7B** has the same KV cache as Mistral 7B. Only the active expert weights differ, not the attention architecture.
- **Qwen 2.5 32B at 32K** eats 8GB of KV cache. With Q4 weights (~20GB) and overhead, you need about 29GB total. It fits on a 4090 but just barely.

For any model, doubling context length doubles KV cache VRAM. And for larger models, the KV cache grows with layer count, not parameter count. Llama 3.1 70B has 80 layers vs. 32 for the 8B, so its KV cache is 2.5x larger (not 8.75x larger) because both use 8 KV heads.

---

6 ways to reduce KV cache VRAM

1. Use the context length you actually need

This is the most effective optimization and it costs you nothing. If you're doing Q&A, chatbot conversations, or coding assistance, you almost certainly don't need 32K+ context. Most conversations fit in 4K-8K tokens.

```
# llama.cpp: set context to 4096 instead of model maximum
llama-server -m model.gguf -ngl 99 --ctx-size 4096

# Ollama: set in Modelfile
PARAMETER num_ctx 4096
```

The savings are dramatic. For Llama 3.1 8B:

Context	KV Cache (FP16)	Total VRAM (Q4 weights)
2,048	0.25 GB	~5.7 GB
4,096	0.5 GB	~5.9 GB
8,192	1.0 GB	~6.4 GB
32,768	4.0 GB	~9.4 GB
131,072	16.0 GB	~21.4 GB

Going from 32K to 4K saves 3.5GB. That's the difference between fitting on an 8GB card and not fitting. Don't set 128K context "just in case." Set it to what your workload actually requires.

2. Q8 KV cache quantization

Instead of storing KV vectors in FP16 (2 bytes per value), you can quantize them to 8-bit (1 byte per value). This halves the KV cache size with minimal quality loss. Most benchmarks show negligible degradation.

In llama.cpp:

```
llama-server -m model.gguf -ngl 99 --ctx-size 32768 \
    --cache-type-k q8_0 --cache-type-v q8_0
```

In Ollama (requires setting via Modelfile or environment):

```
# In your Modelfile
PARAMETER kv_cache_type q8_0
```

With Q8 KV cache, Llama 3.1 8B at 32K context drops from 4.0GB to **2.0GB** of KV cache. Combined with Q4 model weights, total VRAM goes from ~9.4GB to ~7.4GB. That's the difference between needing 12GB and fitting on 8GB.

This is the best bang-for-buck optimization after reducing context length. Do it.

3. Q4 KV cache quantization

You can push further and quantize the KV cache to 4-bit (0.5 bytes per value). This cuts KV cache to 25% of FP16.

```
llama-server -m model.gguf -ngl 99 --ctx-size 32768 \
    --cache-type-k q4_0 --cache-type-v q4_0
```

At 32K context, Llama 3.1 8B KV cache drops from 4.0GB (FP16) to **1.0GB** (Q4). Massive savings.

The catch: Q4 KV cache can degrade quality, especially on long contexts where attention precision matters most. Short conversations are fine. Long document analysis or multi-step reasoning over many pages — you might notice degradation. Test it with your workload before committing.

A good middle ground: Q8 keys with Q4 values. Keys need more precision than values in practice:

```
llama-server -m model.gguf -ngl 99 --ctx-size 32768 \
    --cache-type-k q8_0 --cache-type-v q4_0
```

4. Flash attention

Flash Attention doesn't reduce KV cache size. The same Key/Value vectors still need to be stored. What it does is compute attention more efficiently by breaking the computation into smaller blocks that fit in GPU SRAM, avoiding repeated reads from slower VRAM.

The result: faster inference and lower peak memory usage during the attention computation itself. It won't save you from a 16GB KV cache, but it'll make operations on that cache faster and avoid extra temporary memory allocation.

```
# llama.cpp: enable flash attention
llama-server -m model.gguf -ngl 99 --ctx-size 32768 --flash-attn
```

Flash Attention is enabled by default in most modern inference backends. In llama.cpp, pass `--flash-attn` (or `-fa`) explicitly. In vLLM, it's on by default. In Ollama, it's handled automatically.

The real benefit is at long contexts (16K+) where attention computation is the bottleneck. At 4K context, you won't notice much difference.

5. Sliding window attention

Some models skip storing the full context in KV cache entirely. Sliding Window Attention (SWA) keeps only the last N tokens in cache and discards older ones.

Models using SWA:

- **Mistral 7B**: 4,096-token sliding window
- **Gemma 3**: Hybrid with sliding window layers (uses 1,024-token window on most layers, full attention on every 6th layer)
- **Mixtral 8x7B**: 4,096-token sliding window

With SWA, the KV cache has a fixed maximum size regardless of how long the conversation goes. Mistral 7B's KV cache never exceeds its 4K window allocation even if you set context to 32K. The model simply can't "see" tokens older than 4K.

The tradeoff is real: the model genuinely cannot reference information from beyond the window. The attention mechanism cannot attend to those older tokens at all. For conversational use, this rarely matters. For document analysis where you need the model to reference page 1 while reading page 50, it will hurt.

You don't configure SWA. It's baked into the model architecture. But you should know which models use it when picking what to run, because it directly affects how much VRAM context costs you.

6. Context pruning in frontends

If you use Open WebUI or SillyTavern, both can manage context for you:

- **Open WebUI**: Automatically truncates older messages when context fills up. You can configure the max context in the model settings.
- **SillyTavern**: Has configurable context management. It can summarize old messages instead of just dropping them, or use a sliding window approach to keep the most relevant parts of conversation history.

This doesn't reduce the KV cache for a given context length. It reduces how much context you actually send to the model, keeping you within a manageable range. Practical if you want long conversations without manually restarting them.

How to check your actual KV cache usage

llama.cpp verbose output

Run llama.cpp with verbose logging and you'll see exactly how much KV cache is allocated:

```
llama-server -m model.gguf -ngl 99 --ctx-size 32768 --verbose
```

Look for lines like:

```
llm_load_print_meta: n_ctx_train      = 131072
llm_load_print_meta: n_embd_head_k    = 128
llm_load_print_meta: n_embd_head_v    = 128

kv_cache_init:  CUDA0 KV buffer size =  4096.00 MiB
```

That `KV buffer size` line is your answer. It tells you exactly how much VRAM the KV cache reserved. If you see it allocating 4096 MiB but you only have 3GB of headroom, you know why it's failing.

nvidia-smi during inference

```
# Watch VRAM usage in real time
watch -n 1 nvidia-smi
```

Load the model at minimum context, note the VRAM usage, then load it at your target context. The difference is roughly your KV cache allocation. Not exact (there's other overhead), but good enough to see what's going on.

llama.cpp flags reference

These are the flags that control KV cache behavior:

```
# Core context/cache flags
--ctx-size N          # Set context window size (default: 4096)
--cache-type-k TYPE   # KV cache type for keys: f16, q8_0, q4_0 (default: f16)
--cache-type-v TYPE   # KV cache type for values: f16, q8_0, q4_0 (default: f16)
--flash-attn          # Enable flash attention (recommended)

# Useful for debugging
--verbose             # Show detailed memory allocation info
```

How this connects to the planning tool

Our VRAM Planning Tool uses the same formula from this article to estimate total VRAM requirements. When you select a model and context length, the tool computes:

1. **Model weights** at your chosen quantization
2. **KV cache** using the formula: `batch × ctx × 2 × layers × kv_heads × head_dim × bytes_per_param`
3. **Overhead** (~500MB for CUDA context and framework)

If the tool tells you a configuration doesn't fit, now you know exactly where the VRAM is going. You can decide whether to shrink the model, reduce context, or quantize the KV cache.

What to do about it

My recommended order of operations for managing KV cache VRAM:

1. **First**: Set context to what you actually need. 4K-8K covers most use cases. Don't set 128K by default.
2. **Second**: Enable Q8 KV cache (`--cache-type-k q8_0 --cache-type-v q8_0`). Halves KV size, no meaningful quality loss.
3. **Third**: If still tight, try Q4 KV cache for values and Q8 for keys. Test quality with your workload.
4. **Fourth**: Enable flash attention for faster computation at long contexts.
5. **If nothing works**: You need either a smaller model, a shorter context, or more VRAM.

The KV cache is not something to fear. It's something to budget for. If you know the formula and you know your model's architecture, you can predict exactly how much VRAM any configuration will need before you download a single file.

That 128K context window on the model card? It's a ceiling, not a recommendation. Treat it that way and you'll stop running into CUDA out of memory errors wondering what went wrong.

Related guides

- Context Length Explained: Why It Eats Your VRAM: broader overview of context windows and practical limits per VRAM tier
- How Much VRAM Do You Need for Local LLMs?: full VRAM breakdown by model size and quantization
- Context Length Exceeded: What To Do When Your Model Runs Out of Space: troubleshooting when context fills up
- Why Is My Local LLM So Slow?: memory bandwidth, CPU offloading, and other performance bottlenecks

Source: https://insiderllm.com/guides/kv-cache-optimization-guide/

Free guides for running AI locally