Why Is My Local LLM So Slow? A Diagnostic Guide

February 18, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** The most common cause is CPU inference — your model is running on your processor instead of your GPU. Run nvidia-smi while generating. If GPU utilization is 0%, Ollama or your backend isn't detecting your GPU. Second most common: the model is too big for your VRAM, so layers are offloading to system RAM at 10-15x slower bandwidth. Third: context length is set too high, eating VRAM and forcing offload. Fix these three and you'll solve 90% of slow inference.

📚 **More on this topic:** VRAM Requirements · llama.cpp vs Ollama vs vLLM · CPU-Only LLMs · Mac Mini M4 for Local AI · Planning Tool

Your local model is generating tokens like it's thinking really hard about each one. ChatGPT streams instantly. Your local setup crawls. Something is wrong — but what?

This is a diagnostic guide. Work through the checks in order. Most people find their problem in the first three.

What "Good" Speed Looks Like

Before diagnosing, know what to expect. These are realistic token generation speeds for models that fit entirely in GPU VRAM:

Hardware	Model	Expected Speed
8GB VRAM (RTX 3060 Ti, 4060)	7B Q4_K_M	30-50 tok/s
12GB VRAM (RTX 3060 12GB)	14B Q4_K_M	20-30 tok/s
16GB VRAM (RTX 4060 Ti 16GB)	14B Q4_K_M	25-40 tok/s
24GB VRAM (RTX 3090, 4090)	32B Q4_K_M	15-35 tok/s
CPU only, 16GB RAM	7B Q4_K_M	3-8 tok/s
M4 Pro 48GB	32B Q4_K_M	15-22 tok/s

If you're getting numbers in these ranges, your setup is working correctly. ChatGPT feels faster because it runs on datacenter hardware with 80GB GPUs. Local inference on consumer hardware is slower — that's the tradeoff for privacy and zero cost per query.

If you're getting significantly below these numbers, keep reading.

Check 1: Is the Model Running on GPU or CPU?

This is the #1 cause of slow local inference.

Open a terminal and run while your model is generating:

```
nvidia-smi
```

Look at two things:

- **GPU-Util**: Should be 30-90% during generation. If it's 0%, your model is running on CPU.
- **Memory-Usage**: Should show your model loaded. If it's near 0, nothing is on the GPU.

On Linux, `nvtop` gives a real-time view. On Mac, check Activity Monitor → GPU History.

If GPU utilization is 0%

Your inference engine isn't detecting your GPU. Common causes:

Ollama: Run `ollama ps` — if it shows CPU as the compute type, Ollama can't find CUDA. Fix:

```
# Check if NVIDIA drivers are installed
nvidia-smi

# If nvidia-smi fails, install drivers
# Ubuntu/Debian:
sudo apt install nvidia-driver-535

# After driver install, restart Ollama
sudo systemctl restart ollama
```

llama.cpp: You compiled without CUDA support. Rebuild:

```
cmake -B build -DGGML_CUDA=ON
cmake --build build --config Release
```

Docker: You forgot the `--gpus all` flag:

```
docker run --gpus all -p 11434:11434 ollama/ollama
```

Going from CPU to GPU inference typically means **5-15x faster** generation. This single fix transforms the experience.

Check 2: Is the Model Too Big for Your VRAM?

If part of your model is offloaded to system RAM, you're running at a blend of GPU speed and RAM speed. Even a few offloaded layers cause a noticeable slowdown.

How to tell: In Ollama, watch the logs when loading a model:

```
# Check Ollama logs
journalctl -u ollama -f
# or
ollama run qwen3:8b --verbose
```

If you see messages about offloading layers to CPU or "not enough VRAM," that's your problem.

In llama.cpp, the `--n-gpu-layers` (`-ngl`) flag controls how many layers go to GPU. If you set it lower than the model's total layers, the rest run on CPU.

The speed penalty

Offload Scenario	Typical Speed
100% GPU	30-50 tok/s (7B)
80% GPU, 20% CPU	15-25 tok/s
50% GPU, 50% CPU	8-15 tok/s

Offload Scenario	Typical Speed
100% CPU	3-8 tok/s

Fix: Use a model that fits entirely in VRAM. Check our VRAM requirements guide for exact sizes at every quantization level. If your 14B model doesn't fit, drop to 8B — a fully GPU-loaded 8B model is faster and more responsive than a partially offloaded 14B.

Check 3: What Quantization Are You Using?

Quantization level affects both size and speed. The relationship isn't always obvious:

Quantization	VRAM (7B)	Speed	Notes
Q8_0	~8 GB	Fast	Needs more VRAM, but compute is efficient
Q6_K	~6 GB	Fast	Great balance
Q4_K_M	**~5 GB**	**Fastest for most GPUs**	**Sweet spot**
Q3_K_S	~3.5 GB	Slower on some hardware	Smaller but more complex dequant
Q2_K	~2.5 GB	Often slower than Q3	Heavy quality loss AND speed loss

Q4_K_M is typically the fastest quantization level because it balances memory bandwidth (less data to read) with dequantization overhead (less math to unpack). Going lower than Q4 doesn't always make things faster — the dequantization overhead can outweigh the bandwidth savings.

Fix: If you're running Q2 or Q3 and it's slower than expected, try Q4_K_M. It might actually be faster while also producing better quality output.

Check 4: Context Length Too High?

The KV cache grows with context length and eats VRAM. If your context setting is pushing VRAM usage to the edge, layers start offloading to RAM and everything slows down.

Default context lengths are often too generous:

- Ollama defaults vary by model — some set 8K, some set 32K
- llama.cpp defaults to 2048 unless you specify `-c`

- LM Studio may default to the model's maximum (sometimes 128K)

The symptoms: Model loads fine, first few messages are fast, then it gradually slows down as the conversation gets longer and the KV cache grows.

Fix: Set context to what you actually need:

```
# llama.cpp: 4096 is plenty for most conversations
llama-cli -m model.gguf -ngl 99 -c 4096

# Ollama: set in Modelfile
FROM qwen3:8b
PARAMETER num_ctx 4096
```

For quick Q&A, 2048 is enough. For longer conversations, 4096-8192 is comfortable. Only set 32K+ if you're specifically doing long-document work and have the VRAM to spare.

Check 5: Are You Using the Right Backend?

Not all inference engines are equal in speed on the same hardware.

On NVIDIA GPUs: ExLlamaV2 generates tokens 50-85% faster than llama.cpp. If you're using Ollama (which wraps llama.cpp) and speed is your priority, switching to ExLlamaV2 via TabbyAPI is a significant upgrade. The tradeoff: more complex setup, NVIDIA-only, model must fit entirely in VRAM.

On Apple Silicon: LM Studio with the MLX backend is 30-50% faster than Ollama (llama.cpp) for token generation. If you're on a Mac and using Ollama, try LM Studio with MLX.

Ollama vs raw llama.cpp: Ollama adds a thin server layer over llama.cpp. The overhead is typically 5-10% — not worth switching over unless you need maximum control. But if you're already comfortable with the command line, `llama-server` gives you slightly more speed plus fine-grained control over GPU layers, cache modes, and batch size.

Check 6: RAM Speed (CPU and Apple Silicon)

If you're running on CPU or Apple Silicon, memory bandwidth directly determines your speed. Every token requires reading the entire model weights from memory.

Memory Type	Bandwidth	Impact
DDR4-2133	~17 GB/s	Very slow inference
DDR4-3200	~25 GB/s	Slow
DDR5-4800	~38 GB/s	Moderate
DDR5-6000	~48 GB/s	Decent for CPU inference
Apple M4	120 GB/s	Good
Apple M4 Pro	273 GB/s	Fast

On NVIDIA GPUs, this doesn't matter much — VRAM bandwidth (936 GB/s on RTX 3090) dwarfs system RAM. But for CPU inference or Apple Silicon, check your RAM speed in BIOS or System Information. Upgrading from DDR4-2133 to DDR4-3200 (often just an XMP profile toggle) can improve CPU inference by 30-40%.

Check 7: Thermal Throttling

Laptops especially. If your GPU or CPU hits its thermal limit, it clocks down to protect itself. You'll see decent speed for the first minute, then a gradual slowdown.

How to tell: Watch GPU temperature in `nvidia-smi` or `nvtop`. If it hits 83-90°C and stays there, you're throttling.

Fixes:

- Elevate the laptop — even a book under the back edge helps airflow
- Use a cooling pad
- Clean dust from vents (the #1 free fix for old laptops)
- In summer, ambient temperature matters — a 30°C room limits cooling headroom

For desktops, check that GPU fans are spinning and case airflow isn't blocked. A GPU thermal pad replacement on used cards (especially used 3090s) can drop temperatures 10-15°C.

Quick Diagnostic Flowchart

1. Run `nvidia-smi` during generation → GPU util 0%? → **GPU not detected.** Fix drivers/CUDA.
2. GPU active but slow? → Check if model fits in VRAM → Layers offloaded? → **Use smaller model or lower quant.**
3. Model fits, still slow? → Check context length → Set too high? → **Reduce to 4096.**
4. Context is reasonable? → Check backend → Try ExLlamaV2 (NVIDIA) or MLX (Mac).
5. Already optimal backend? → Check thermals → Throttling? → **Improve cooling.**
6. Everything checks out? → Your speed is probably normal. Check the expectations table above.

Bottom Line

90% of slow local LLM performance comes from three causes: CPU inference (GPU not detected), partial VRAM offloading (model too big), or excessive context length. Fix those first. The remaining 10% is backend choice, RAM speed, and thermals.

If your hardware is working correctly and you're hitting the expected speeds above, that's as fast as it gets without upgrading. The next meaningful speed jump comes from more VRAM — and the best value upgrade is still a used RTX 3090 at $700-900 for 24GB.

Source: https://insiderllm.com/guides/why-local-llm-slow/