LLM Running Slow? Two Different Problems, Two Different Fixes

March 5, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** Local LLM slowness is actually two different problems. Slow time-to-first-token (the pause before text starts) is compute-bound — fix it with Flash Attention (OLLAMA_FLASH_ATTENTION=1), smaller batch sizes if VRAM is tight, and shorter prompts. Slow generation speed (low tok/s once text is flowing) is memory-bandwidth-bound — fix it by ensuring the model runs 100% on GPU (check with nvidia-smi), using Q4_K_M quantization, and reducing context length. Most people conflate these. Fixing the wrong one wastes your time.

📚 **More on this topic:** VRAM Requirements · llama.cpp vs Ollama vs vLLM · Ollama Not Using GPU · Why Is My Local LLM So Slow?

You type a prompt, hit enter, and… nothing. The cursor blinks. Three seconds pass. Five. Then text starts trickling out, one word at a time, slower than you can read.

That frustration is actually two separate problems that most guides mash together. The long wait before any text appears and the slow trickle once it starts have different causes and different fixes. I spent weeks tuning the wrong knobs before I figured this out.

The Two Phases of LLM Inference

Every time you send a prompt, your model runs two distinct operations:

Phase 1: Prefill (prompt processing) The model reads your entire prompt, processes every token in it, and builds the KV cache. This is compute-bound — your GPU's raw processing power matters here. The result you feel: the pause between hitting enter and seeing the first token appear. This is your **time-to-first-token (TTFT)**.

Phase 2: Decode (token generation) The model generates output tokens one at a time, reading the full model weights from memory for each token. This is memory-bandwidth-bound — how fast your GPU can read data from VRAM matters here. The result you feel: the speed at which text streams out. This is your **tokens per second (tok/s)**.

Two different bottlenecks. Two different fixes.

Metric	Phase	Bottleneck	What You Feel
TTFT	Prefill	GPU compute	Wait before text starts
tok/s	Decode	Memory bandwidth	Speed of text streaming

What “Normal” Looks Like

Before you start tuning, know whether you actually have a problem. These are realistic numbers for a 7-9B model at Q4_K_M with a short prompt (~100 tokens) and 4K context:

GPU	TTFT	Generation (tok/s)
RTX 4090 (24GB)	<0.5s	70-90
RTX 3090 (24GB)	<0.5s	40-55
RTX 4070 Ti Super (16GB)	<0.5s	35-50
RTX 4060 Ti 16GB	<1s	15-25
RTX 3060 12GB	<1s	12-20
RTX 4060 8GB	<1s	10-18
M4 Pro (48GB)	<1s	20-30
CPU only (DDR5)	3-8s	3-8

For 27-32B models at Q4, divide the tok/s roughly by 3-4 and multiply TTFT by 2-3. For 70B at Q4, divide tok/s by 6-8.

If your numbers are in these ranges, your setup is working correctly. The hardware is the limit, not the software. If you’re well below these, keep reading.

Fixing Slow Time-to-First-Token

A long TTFT means the prefill phase is taking too long. Your prompt is being processed slowly.

Check 1: Is the Model on GPU?

If TTFT is measured in many seconds for a short prompt, the model is probably on CPU. Run `nvidia-smi` during inference. If GPU utilization is 0%, see our Ollama not using GPU guide — that's your whole problem.

Check 2: Enable Flash Attention

Flash Attention restructures how the attention mechanism reads and writes memory during prefill. It speeds up TTFT without affecting output quality at all.

Ollama:

```
# Set as environment variable before starting Ollama
OLLAMA_FLASH_ATTENTION=1 ollama serve

# Or for systemd service
sudo systemctl edit ollama
# Add: Environment="OLLAMA_FLASH_ATTENTION=1"
sudo systemctl restart ollama
```

llama.cpp:

```
llama-server -m model.gguf -ngl 99 -fa
# -fa enables Flash Attention
```

On long prompts (2K+ tokens), Flash Attention can cut TTFT by 30-50%. On short prompts you won't notice much difference, but there's no reason not to enable it.

Check 3: Prompt Length

TTFT scales with prompt length. The model must process every token in your input before generating the first output token. A 100-token prompt processes fast. A 4,000-token prompt with a long system prompt, conversation history, and RAG context takes much longer.

Fixes:

- Trim your system prompt. Most are padded with instructions the model ignores anyway.

- Reduce conversation history. Ollama sends the full conversation each turn by default. Long conversations balloon TTFT.
- If you're doing RAG, send fewer but more relevant chunks instead of dumping everything.

Check 4: Batch Size (Advanced)

During prefill, the model processes prompt tokens in batches. The `n_batch` parameter controls how many tokens are processed per step.

llama.cpp:

```
# Default is 2048. Increase for faster prefill if you have VRAM headroom:
llama-server -m model.gguf -ngl 99 -fa -b 4096

# If VRAM is tight, lower it:
llama-server -m model.gguf -ngl 99 -fa -b 512
```

Larger batch = faster prefill but more VRAM used during the prefill phase. If you're already close to your VRAM limit, a large batch can push layers to CPU and make everything worse. Match batch size to your available headroom.

Ollama doesn't expose batch size directly, but Flash Attention gives you most of the prefill benefit without needing to tune this.

Fixing Slow Token Generation (Low tok/s)

Once text starts streaming, slow generation is almost always a memory bandwidth problem. The GPU reads the entire model weights from VRAM for every single token it generates. More data to read per token = slower. Less bandwidth available = slower. That's the whole story.

Fix 1: Make Sure the Full Model Is on GPU

This matters more than everything else combined. If even a few layers are offloaded to system RAM, generation speed tanks. GPU VRAM bandwidth on an RTX 3090/4090 is 900-1,000 GB/s. Your system DDR5? 40-50 GB/s. A 20x gap.

Check in Ollama:

```
ollama ps
# Look for "100% GPU" vs "60% GPU/40% CPU"
```

Check in llama.cpp:

```
# Set -ngl high enough to load ALL layers
llama-server -m model.gguf -ngl 99
# If you see "offloaded X/Y layers" in the log, you need more VRAM or a smaller model
```

A 7B model fully on a slow GPU will outperform a 14B model split between GPU and CPU. If your model doesn't fit, drop to a smaller size or lower quantization before accepting partial offload.

Fix 2: Use Q4_K_M Quantization

Quantization directly affects generation speed because it determines how much data the GPU must read per token.

Quantization	Data per Token (7B)	Relative Speed
FP16	~14 GB	Baseline (1x)
Q8_0	~7 GB	~2x faster
Q6_K	~5.5 GB	~2.5x faster
Q4_K_M	~4 GB	~3.5x faster
Q3_K_S	~3 GB	~3x faster*

*Q3 and below can actually be slower than Q4 due to dequantization overhead. Q4_K_M is the sweet spot: less data to read, minimal dequant cost.

If you're running FP16 or Q8 and your tok/s is low, switching to Q4_K_M will give a large, immediate speedup.

Fix 3: Reduce Context Length

Context length doesn't just eat VRAM, it slows generation too. The KV cache that stores your conversation state gets read during each token generation step. A 32K context builds a much larger cache than 4K.

```
# llama.cpp: set to what you actually need
llama-server -m model.gguf -ngl 99 -c 4096

# Ollama Modelfile:
FROM qwen3.5:9b
PARAMETER num_ctx 4096
```

Each doubling of context length costs roughly 10-20% generation speed. If your tool defaults to 32K or 128K and you're having a short conversation, you're paying a speed tax for context you aren't using. Set it to 4096 for general chat, 8192 if you need longer conversations, and only go higher when you specifically need it.

Fix 4: Keep the Model Loaded

If the first response after a while is slow but subsequent ones are fast, the model is being unloaded from VRAM between requests.

Ollama:

```
# Default unloads after 5 minutes. Keep loaded longer:
OLLAMA_KEEP_ALIVE=24h ollama serve

# Or keep loaded indefinitely:
OLLAMA_KEEP_ALIVE=-1 ollama serve
```

Model loading takes 2-10 seconds depending on size. If you're using the model throughout the day, keep it resident.

Fix 5: Threads (CPU and Hybrid Inference)

If layers are running on CPU (because the model doesn't fully fit in VRAM, or you're on CPU-only), thread count matters.

llama.cpp:

```
# Set threads to your physical core count (not logical/hyperthreaded)
llama-server -m model.gguf -ngl 99 -t 8
```

More threads than physical cores usually hurts. Hyperthreading doesn't help LLM inference because the workload is memory-bound, and the extra threads just compete for bandwidth. Check your physical core count with `lscpu` on Linux or `sysctl -n hw.physicalcpu` on Mac.

The Tuning Cheat Sheet

Sorted by impact:

Problem	Fix	Impact	Effort
Model on CPU	Install GPU drivers, rebuild with CUDA	5-15x speedup	Medium
Partial VRAM offload	Smaller model or lower quant	2-5x speedup	Easy
No Flash Attention	`OLLAMA_FLASH_ATTENTION=1` or `-fa`	30-50% faster TTFT	Easy
Wrong quantization	Switch to Q4_K_M	2-3x faster generation	Easy
Context too high	Reduce `num_ctx` / `-c` to 4096	10-30% faster generation	Easy
Model unloading	`OLLAMA_KEEP_ALIVE=-1`	Eliminates reload delay	Easy
Wrong thread count	`-t` = physical cores	10-30% faster CPU layers	Easy
Large batch on tight VRAM	Lower `-b` to 512	Faster TTFT, avoids offload	Easy

---

When the Hardware Is the Limit

After all this tuning, you might find your speed already matches the benchmarks above. If so, your software is doing everything right. The hardware is the bottleneck, and no flag or config change will fix that.

Generation speed is capped by memory bandwidth. Here's the math: a 7B Q4 model has roughly 4GB of weights. At 55 tok/s, the GPU reads 4GB x 55 = 220 GB/s. An RTX 3090 has 936 GB/s bandwidth. Some of that goes to KV cache reads, framework overhead, and cache misses. The numbers check out. You're using the hardware correctly.

To go faster, you need more bandwidth:

- RTX 3090 to RTX 4090: ~25-40% faster (more bandwidth, newer architecture)

- RTX 4090 to RTX 5090: ~30-50% faster (GDDR7 bandwidth jump to 1,792 GB/s)
- Any discrete GPU to Apple M5 Max: you trade peak speed for capacity. Lower tok/s per model size, but you can fit much bigger models

For specific GPU recommendations, check our VRAM requirements guide and GPU buying guide.

Bottom Line

Slow TTFT and slow tok/s are different problems. If the wait before text starts is the issue, enable Flash Attention, shorten your prompts, and tune batch size. If the text itself streams slowly, make sure the model is 100% on GPU, use Q4_K_M, and reduce context length. Fix the right problem and you'll stop wasting time on changes that don't help.

Related Guides

- VRAM Requirements for Every Local LLM
- llama.cpp vs Ollama vs vLLM: When to Use Each
- Ollama Not Using GPU: Complete Fix Guide
- Why Is My Local LLM So Slow?
- Quantization Explained

Get notified when we publish new guides.

Subscribe — free, no spam
