

Strata vs llama.cpp on a 3060: The 7x Is Real. Two Flags Halve It

October 5, 2026 · by Mark Bartlett

[Download this guide as PDF](#)

Quick Answer: On a 12 GB card, a 125B MoE takes minutes to read a long prompt, and an agent feeds it long prompts all day. Strata, a two-week-old engine built for one model, read a 32K prompt in 30 seconds on my RTX 3060, where stock llama.cpp took three and a half minutes. Most of that gap is a llama.cpp setting you can change tonight. What's left after it is still real, and it shows up in decode too.

 **More on this topic:** [A 177B Model on a 3060: The 32 GB Number](#) · [Every SSD-Streaming MoE Engine](#) · [MoE Models Explained](#) · [What Can You Run on 12GB VRAM](#) · [InsiderLLM Benchmarks](#)

Run a 125B mixture-of-experts model on a 12 GB card and decode is tolerable. Reading the prompt is what hurts — chat sends a few hundred tokens per turn and you barely notice. A coding agent sends the repo map, the open files and the tool output, 20,000 or 30,000 tokens at a time, every turn — and on a 3060 with stock llama.cpp that is minutes of staring at nothing before the first token arrives.

[Codacus put a number on a fix](#). In his video, [Strata](#), an open-source inference engine that runs exactly one model family, Qwen3.8-Flash-Next, prefilled about 7.5x faster than llama.cpp on his machine. His box is not the box most people own. I tested the same claim on a cheaper one: an RTX 3060 12 GB on PCIe 3.0, an i7-7700 and 32 GB of DDR4-2133, the machine a used 3060 usually ships in. Pre-registered, same files for both engines, three measured requests per cell, swap counted on every one.

The short version

prompt	stock llama.cpp	llama.cpp, -b 4096 -ub 4096	Strata v0.1.39	Strata ÷ tuned llama.cpp
4,096 tokens	158 tok/s	399–405	933	2.3x
16,384	160	471–485	1,025	2.1–2.2x
32,512	158	416–441	1,094	2.5–2.6x

Prefill, mean of three measured requests. The middle column is two llama.cpp arms, plain and with pinned host memory; their ranges overlap, so I show both ends. In every cell, Strata's min-max range clears llama.cpp's, so "faster" here means faster, not noise.

What that means in seconds, for the 32K prompt an agent sends: stock llama.cpp took **208 s** to the first token, tuned llama.cpp **75–80 s**, Strata **30 s**.

Bar chart: time to first token on a 32,512-token prompt on an RTX 3060. Stock llama.cpp 208 seconds, llama.cpp with -b 4096 -ub 4096 80 seconds, the same plus pinned memory 75 seconds, Strata 30 seconds.

Four findings, in the order they matter.

1. The 7x reproduces against stock llama.cpp

Stock llama.cpp at default batch settings prefilled at 158 tok/s at every length. Strata did 933 at 4K, 1,025 at 16K and 1,094 at 32K. That is **5.9x**, **6.4x** and **6.9x**. Codacus's 7.5x was on his hardware; on a four-core box with a gen-3 bus, the top end lands at 6.9x. Close enough that I'd call it reproduced.

The model is Qwen3.8-Flash-Next, Qwen's preview of its Qwen 4 architecture, and the file both engines ran is [ISTA-DASLab's Coder IQ1_M](#): a code-specialised cut that keeps 256 of the 512 experts per layer — 58.4 GB in two shards. On a 12 GB card most of those experts live in system RAM. For every slice of the prompt it processes, the GPU needs every expert that slice routes to, and llama.cpp's default slice, the micro-batch, is 512 tokens. A 32K prompt is 64 slices — 64 trips of the offloaded experts across the PCIe bus.

Strata reads the prompt in chunks of 8,192 tokens and streams the experts it doesn't hold through a ring, so the next layer's arrive while the current layer computes. Four chunks, four trips.

2. One batch setting closes most of the gap, and it isn't Strata's

Raise llama.cpp's micro-batch to 4,096 and the trip count drops from 64 to 8. Here is the exact command I ran:

```
llama-server -m Qwen3.8-Flash-Next-GSQ-RC0-IQ1_M-00001-of-00002.gguf \
  -ngl 99 -ncmoe 44 -fa on -c 34816 -np 1 --lazy-mode on -t 4 \
  -b 4096 -ub 4096
```

The gotcha — it's two flags, not one. llama.cpp caps `-ub` at `-b` (`n_ubatch = min(n_batch, n_ubatch)` in `llama-context.cpp`), and `-b` defaults to 2,048, so `-ub 4096` on its own quietly runs at 2,048. I didn't test that middle setting.

prompt	default -ub 512	-b 4096 -ub 4096	gain
4,096	157.9 (157.6–158.0)	398.7 (361.9–424.6)	2.5x
16,384	159.7 (156.2–161.5)	470.6 (449.7–492.9)	2.9x
32,512	157.6 (156.4–158.3)	416.2 (392.1–433.1)	2.6x

On the 32K prompt, the wait to first token fell from 208 s to 80. Measured as waiting time, that is about three quarters of the distance to Strata, from a setting you can change tonight on any MoE you offload with `-ncmoe`. If you take one thing from this page, take this one.

The trade is VRAM. A 4,096-token micro-batch needs a bigger compute buffer: at `-ncmoe 44` the card held 8.3–8.6 GB with the default and 10.9 GB with the big batch. I picked 44 as the lowest value at which the big-batch config both loaded and served a 32K prompt; at 42 it loaded and then failed on the first long request. If you've already tuned `-ncmoe` to fill the card at the default batch, expect to raise it by a couple of layers to make room.

The third llama.cpp arm added `GGML_CUDA_REGISTER_HOST=1`, which pins the host copy of the experts. On this box it gave +1–6% on the means, inside the noise at every length. The batch size is the lever — pinning isn't.

The bus traffic shows why. With the default batch, the card pulled 6.1–7.2 GB/s over PCIe during prefill. With the big batch, 0.7–3.9 GB/s, and it ran 2.6x faster. The default config kept the bus busier and got less done — it was re-sending the same experts every 512 tokens.

3. What remains is real: 2.1–2.6x on prefill, 2.3–2.5x on decode

Tuned llama.cpp is still slower than Strata at every length — and not by a little: **2.3x at 4K, 2.1–2.2x at 16K, 2.5–2.6x at 32K**. The ranges don't overlap anywhere.

I also ran Strata's own chunk size down to see where its edge comes from. Forced to 2,048-token chunks on the 32K prompt it prefilled at 802 tok/s; at 4,096, 987; at 8,192 (what its automatic setting picked on this card), 1,096. Same direction as llama.cpp's batch effect — part of Strata's lead is simply that it defaults to big chunks. But at 2,048-token chunks it's still nearly 2x tuned llama.cpp at 4,096-token batches, and the bus never ran flat out: 24% of its measured 13.9 GB/s at 8K chunks, 59% at 2K. On this box the bus is part of the story, not the limit.

Decode needs more care, because Strata ships with drafting on. It uses the model's own multi-token-prediction layer to guess ahead, plus prompt lookup, which drafts from text the reply is repeating out of the prompt. llama.cpp in these runs had no drafter at all. Put those two numbers side by side and you're measuring the drafter as much as the engine.

Strata won't run without its drafting machinery on this file. `--spec 0` is refused at startup, and the server won't start without the draft layer loaded. So the engine-speed arm caps the draft window at zero tokens (`--spec 2 --mtp-max-t 1`): draft layer loaded — nothing drafted. All nine measured requests reported zero drafts.

prompt	Strata as shipped (drafting + prompt lookup)	Strata, no drafts (engine speed)	llama.cpp, no drafter
4,096	30.2 tok/s	21.1 (17.2–23.4)	9.0–9.1
16,384	30.9	20.1 (18.6–22.7)	8.5–8.6
32,512	30.6	19.7 (17.3–21.0)	7.9–8.0

The middle column is the like-for-like number: **2.3x, 2.3x and 2.5x** llama.cpp's best arm at each length. Strata's slowest no-drafts request (17.2) is still nearly twice llama.cpp's fastest (9.2). Expect variation, though — the no-drafts spread runs up to 30% within a cell, where drafting-on stays under 8%.

The left column is what you'll actually see if you install it and change nothing: about 30 tok/s, with roughly 72% of drafted tokens accepted on this code-summary task. Drafting adds 43–55% over the engine number. How much it adds on your prompts depends on how predictable your output is, and code that quotes its input is the best case.

4. On 32 GB, Strata swapped nothing out

I counted swap on every measured request, in and out, because a [re-run of the September Flash-Next config](#) on this same box showed it swapping during measurement, which September never recorded.

Strata swapped out **0 KiB on every measured request**, all 27 of them across its five runs, with swap-in between 0 and 5 MiB per request. Its first load did push about 380 MB to swap, and later loads moved almost nothing.

llama.cpp swapped out up to **460 MiB in a single request**, and swap in use peaked at 2.4 GB during the default-batch run. That's the page cache at work — llama.cpp maps the 58 GB file instead of owning it, and on 32 GB the kernel juggles that mapping against everything else. The

swap volume didn't track speed request by request. The two slowest 32K llama.cpp requests lined up with the biggest SSD reads in the run, 523 and 660 MiB.

The flip side — and it's a big one — Strata **pins about 22 GB of RAM** (21.7 GiB of experts, page-locked) and leaves roughly **6 GB available** for everything else on the machine. llama.cpp left 22–29 GB available, because mapped file pages count as reclaimable. Strata on a 32 GB box is the only thing that box is doing — close the browser.

Codacus's box vs mine

From his [pinned comment](#), his setup against mine:

	Codacus	Mine
Card	RTX 3060 12 GB	RTX 3060 12 GB
Bus	PCIe 4.0 x8, 13.4 GB/s measured	PCIe 3.0 x16, 13.9 GB/s measured
CPU / RAM	Ryzen 9 7900X, 64 GB DDR5-5600	i7-7700, 32 GB DDR4-2133
Model file	full Qwen3.8-Flash-Next, IQ3_XXS	Coder with half the experts, IQ1_M
Strata version	v0.1.36	v0.1.39
Prefill	926–964 tok/s	933–1,094 tok/s
Decode, as shipped	40–43 tok/s	~30 tok/s

Two readings, not measurements — the file, prompts and Strata version all differ. His bus measured within 4% of mine, and his prefill lands about where mine did with twice the experts per layer. That fits the same 3060 doing the same work per token, since my bus ran at only 27–41% during the stock Strata runs' prefill. His decode beats my 30, which fits CPU and RAM speed driving Strata's decode.

What I'd do

Running any MoE with `-ncmoe` on a 12 GB card: set `-b 4096 -ub 4096` and give back a couple of expert layers to make room. On this model it was a 2.6x prefill gain for free. That holds whether or not Strata is ever on your machine.

Running Qwen3.8-Flash-Next for agents, on Linux, with 32 GB: Strata is worth the setup, if the Coder cut does your job and the machine can be dedicated to it. 30 seconds versus 75 to read a 32K prompt is the difference between an agent you wait on and one you leave running.

Anything else: wait — Strata is two weeks old and changes daily, and the limits below are real.

Limits

- **Speed only.** I didn't measure answer quality. The Coder cut drops half the experts; its authors report 91.3% of the full model's SWE-bench Verified and 98.7% of its LiveCodeBench v6, and Strata's docs say it's weaker outside code and on CJK text. Those are their numbers, not mine.
- **The Coder IQ1_M is the only variant that fits 32 GB without streaming experts off the SSD.** Q2_0 and the larger cuts get mapped from disk on this box, and that's a disk test, not an engine test.
- **One request at a time.** That's Strata's default; `"parallel": 2` exists, and its own README says it makes each answer slower on a 12 GB card.
- **It pins ~22 GB of RAM** while loaded (above), plus a 23 GB `experts.bin` and a 0.9 GB draft layer on disk beside the 58 GB model.
- **Linux means compiling.** Every Strata release so far ships Windows engines only. On Linux, setup builds the engine from source and needs the CUDA 13 toolkit; I installed 13.0.2 in my home directory without root, side by side with the 12.8 my llama.cpp builds use.
- **Output isn't byte-repeatable by default.** At temperature 0, the same prompt can end in a different answer when the drafting or the cache state differs. Strata's docs list opt-in flags for byte-identical repeats; I ran without them.
- **v0.1.39, commit 6f32ec0** . The repo was created on September 24 and had 39 tagged releases eleven days later. Numbers from a later version may differ.
- **The "experimental speed projection" is not a speed feature.** Setup offers it with that name — it's a control vector that projects the refusal direction out of the model, and the repo says so itself, measuring it 0.2–0.4% slower per token. It was off in every run here.
- **License:** the base model is under the **Qwen Community License 1.0**. The ISTA repo is tagged `apache-2.0` ; a quantisation can't relicense its base model. Strata's own code is MIT.
- **One box, one day.** Gen-3 PCIe, a four-core CPU, DDR4-2133. On a gen-4 board, a faster CPU or more RAM, the ratios will move; [Codacus's box](#) is one example. My 3090 box is next, at DDR4-2133 as well.

Method

Hardware. RTX 3060 12 GB, PCIe 3.0 x16 (gen 3 x16 confirmed during every prefill), i7-7700 4c/8t, 32 GB DDR4-2133, SSSTC CA6 256 GB NVMe, Ubuntu 24.04.5, kernel 6.8.0-136, driver 580.173.02, headless. Ollama stopped; the mycoSwarm daemon left running and noted.

File. ISTA-DASLab/Qwen3.8-Flash-Next-GSQ-RCO-Coder-GGUF at revision 5348543e , IQ1_M/ , both shards sha256-checked against the hub: e11083ba...087fad (29,608,446,496 B) and 316b46f3...c161e113 (28,800,138,432 B). GGUF header: qwen4exp , 48 layers, 256 experts, 10 routed per token.

Engines. llama.cpp v0.4.0, commit 5266f24 , CUDA 12.8 bundle, sm_86:

-ngl 99 -ncmoe 44 -fa on -c 34816 -np 1 --lazy-mode on -t 4 , with the defaults (-b 2048 -ub 512), then -b 4096 -ub 4096 , then that plus GGML_CUDA_REGISTER_HOST=1 . Strata v0.1.39, compiled by its setup with nvcc 13.0.88, in the resident low-RAM mode setup picked: --expert-cache auto --prefill auto --spec 4 --spec-min-p 0.5 --mtp ... --max-context 34816 --kv int8 --resident-experts , speed projection off.

Prompts. The llama.cpp v0.4.0 source, cut on the model's own tokenizer to 4,096, 16,384 and 32,512 tokens, each followed by "Summarise what this code does." A fresh random nonce opened every request to defeat prefix caching in both engines; every measured request reported zero cached tokens. Thinking off, temperature 0, 256 output tokens, streamed.

Cells. A discarded priming request before each measured one, three measured per cell; value is the mean, the range is min–max. Prefill and decode are each engine's own reported timings. One server resident at a time.

Memory rule, set before the run. Only MemAvailable under 2 GiB stops a run; swap is allowed and published per request. It never tripped. Lowest MemAvailable: 5.6 GB, with Strata loaded.

Pre-registered expectation. I predicted Strata's prefill before measuring: 600–900 tok/s at 4K, 750–1,150 at 16K, 800–1,200 at 32K. 16K and 32K landed inside. 4K came in 3.7% over the top — the whole 4K prompt fits in one 8,192-token chunk, and I'd charged it for chunk handling it never did.

The amendment. Strata's --spec 0 arm was refused at startup, so the original plan had no drafting-off number. The no-drafts arm was written into the pre-registration and committed before it ran, along with how this page would present decode: as shipped, engine speed and llama.cpp side by side.

The full record (pre-registration, amendments, per-request numbers, swap, PCIe traffic, engine and server logs) is in the [public bench record](#). The same box's [10-05 reference run](#) on the bigger

Unsloth file updated the [Flash-Next article](#)'s stock prefill range. Credit to [Codacus](#) for the video that started this one.

A weekly email with every new guide and measured benchmark.

[Subscribe — free, no spam](#)

Source: <https://insiderllm.com/guides/strata-vs-llama-cpp-rtx-3060/>

Free guides for running AI locally