Gemma 4 26B in 2GB RAM: The MoE Memory Ladder Explained

July 30, 2026

Download this guide as PDF

> **Quick Answer:** TurboFieldfare runs Gemma 4 26B-A4B on an M-series Mac in about 2 GB of RAM by keeping a 1.35 GB shared core resident and streaming the routed experts off the SSD. The headlines call it a new trick. It's the third rung on a ladder, and we've measured the first two on the same model: 128.08 tok/s with everything resident on an RTX 3090, and 37.23 tok/s on a 12GB RTX 3060 with the experts streaming out of DDR4. What makes the SSD tier interesting isn't the speed, it's the floor. Our 3060 stops loading the model at all below -ncmoe 12, where it needs 11.2 GB of VRAM. TurboFieldfare runs the same weights in 2 GB. This walks all three rungs, the boundary where the middle one breaks, and what still hasn't been measured.

📚 **Related:** Best Way to Run Qwen 3.6 35B MoE Locally · MoE Models Explained · Gemma 4 vs Qwen 3.6 on RTX 3090 · Running LLMs on Mac M-Series · Our benchmark dataset

If you saw the headline this week — Gemma 4 26B running in 2 GB of RAM on a MacBook — the useful question isn't whether it's real. It is. The useful question is where it sits relative to what you can already do, because most of the coverage has presented it as a technique arriving from nowhere.

It's the third rung on a ladder. We've measured the first two, on the same model.

What TurboFieldfare actually does

TurboFieldfare is a custom Swift and Metal runtime for Gemma 4 26B-A4B on Apple Silicon. Not a llama.cpp wrapper, not MLX. The author wrote the Metal kernels.

From the repo: a shared core of about **1.35 GB** stays resident along with an FP16 KV cache, the routed experts live on disk and stream in per token, and a **16-slot LFU cache per layer** absorbs repeat hits. Process RSS lands near **2 GB at 4K context** against **~14.3 GB** of weights installed. Quantization is 4-bit MLX affine at group 64 for shared and routed experts, with an 8-bit router.

Source is Apache 2.0; weights come from Hugging Face separately under their own terms. Requirements are narrow: macOS 26 with Metal 4, Xcode 26 and Swift 6.2+, arm64 only, one model at a time, text only.

One detail is getting less attention than it deserves. The repo ships a **103-entry audited experiment record** including, in the author's words, plausible ideas that failed and early results that reversed under stronger validation. Publishing the reversals is rarer than it should be, and it's why I'd weight this repo's numbers above the average launch post.

The ladder, one model, three rungs

An MoE model's experts have to live somewhere. There are three somewheres, and each step down drops roughly an order of magnitude in bandwidth.

Tier	Experts live in	Hardware	Gemma 4 26B-A4B
1	GPU VRAM, fully resident	RTX 3090 24GB	**128.08 tok/s**
2	System RAM, streamed over PCIe	RTX 3060 12GB, DDR4-2133	**37.23 tok/s**
3	SSD, streamed per token	M5 Pro 24GB / M2 Air 8GB	**31-35 / 5.1-6.3**

Tiers 1 and 2 are ours. Tier 3 is the author's, cited and not verified — we own no Mac.

Tier 1 is the May head-to-head: Gemma 4 26B-A4B Q4_K_XL at 128.08 tok/s mean on a single RTX 3090, everything in ~17 GB of VRAM, nothing offloaded anywhere.

Tier 2 is the 3060. Same model family, UD-Q4_K_M at 15.77 GiB, on a card that cannot hold it. `-ngl 99 -ncmoe 12 -fa 1` pins attention and the shared weights on the GPU and pushes the routed experts of 12 layers into system RAM:

Context	Generation	Prompt
empty	37.23 tok/s	674 tok/s
4K	36.09	649
8K	35.75	642

Down 1.5 tok/s across the full 8K. Flat, the way expert streaming usually is, because the per-token cost is constant and flash attention keeps KV growth cheap. All eleven rows, including the failures below, are in the dataset.

The floor of tier 2, which is the whole point

Here's the number that makes the Mac work interesting, and it isn't a speed.

Push the 3060 below `-ncmoe 12` and Gemma 4 doesn't get slower. It stops running.

`-ncmoe`	Generation	Peak VRAM
30 (all experts in RAM)	21.20 tok/s	2,937 MiB
20	27.80 tok/s	7,551 MiB
14	34.50 tok/s	not instrumented
12	**37.23 tok/s**	**11,179 MiB**
8	**fails to load**, rc=1	113 MiB
4	**fails to load**, rc=1	113 MiB
0	**fails to load**, rc=1	113 MiB

Two runs feed that table. The 30 and 20 rows come from a config scan at `-r 2 -p 0`, which is why they carry no prompt-processing figure anywhere in the dataset; 14 and 12 come from a `-r 3` depth run that didn't instrument VRAM, so 12's peak is the scan's figure for the same config.

Read down the curve and it degrades politely: 21 to 28 to 34 to 37 tok/s as you claw experts back onto the card. Then it stops. A 113 MiB peak means the process died before it allocated weights at all. `-ncmoe 12` peaks at 11,179 MiB of the card's 12,288, about **1.1 GB of headroom**, which is why 8 is a wall rather than the next step down. There's nothing left to give back.

So the honest floor for running this model with experts in system RAM, on the most common budget card in local AI, is **11.2 GB of VRAM**.

TurboFieldfare runs the same weights in about **2 GB**.

That's the comparison worth making, and notice it's about memory rather than speed. Going one tier further down, to storage, doesn't buy much throughput — 31-35 on an M5 Pro is in the same range as 37.23 on a 3060. What it buys is the floor dropping by a factor of five.

Why this only works on MoE

Gemma 4 26B-A4B carries 25.2B total parameters and about 3.8B active per token, roughly 15%. TurboFieldfare keeps about 2 GB resident out of 14.3 GB, roughly 14%.

Those fractions being close is the design working, though I'd stop short of calling it a law. The resident set isn't "the active parameters." It's the shared core plus KV cache, a different quantity. Both are small for the same underlying reason: an MoE concentrates most of its weight in experts that mostly don't fire on any given token.

A dense model has no such split. Every parameter is in the hot path every token, so there's nothing to leave on disk, and streaming from storage would mean streaming the whole model per token. That's why "26B in 2 GB" is a headline about MoE, not about compression.

A branding footnote: "A4B" isn't a strict active-parameter count. Google's card says 25.2B total, llama.cpp reports 25.23B, and the name says 26B. Rounded in the flattering direction, and it changes nothing about the technique.

The one comparison here that isn't hand-wavy

Everything above is a ladder across three machines with wide error bars. This next bit isn't, and it's worth separating out.

Push both MoE models we've measured to maximum offload on the same 3060 — every expert in system RAM, nothing left on the card but attention and shared weights:

Model	Total	Active/token	Max offload
Gemma 4 26B-A4B	25.23B	3.8B	**21.20 tok/s**
Qwen 3.6-35B-A3B	34.66B	3B	**28.1 tok/s**

The bigger model is 33% faster.

That reads wrong until you look at the middle column. Once every expert lives in RAM, the thing being measured is how much data crosses the bus per token, and that's set by active parameters. Gemma streams 3.8B per token, Qwen 3B. Total size decides whether it fits at all. Active size decides how fast it runs once it doesn't.

How much weight to put on this: same card, same DDR4-2133, same PCIe 3.0 x16, same llama.cpp build (b10088, commit 67b9b0e), same quantization scheme (UD-Q4_K_M), both at full offload. Two things do differ — the Qwen run used `-r 3` with a prompt sweep, the Gemma run `-r 2` at `-p 0`. So it isn't a single-variable experiment, but it's the closest thing to a controlled result in this article, and a 6.9 tok/s gap is well outside what a repetition-count difference explains.

What the coverage got wrong

One widely-shared writeup states that "standard inference tools like Ollama load all 14.3 GB into memory anyway because fast routing seems to require it."

The reason given there is false. Fast routing does not require full residency, and the proof has been in mainline llama.cpp for months: `--n-cpu-moe` streams only the experts a token actually selects. Tier 2 above is what that looks like measured, on this exact model.

Being precise, because the achievement is real. What's **not** new is streaming routed experts on demand. What **is** new is doing it to **disk** rather than RAM, on Apple Silicon, in a from-scratch Metal runtime with no llama.cpp or MLX underneath, and landing at a usable speed. The framing is what's wrong. The work is fine.

What still hasn't been measured

Three different machines, three different runtimes, and three different quantization schemes — all nominally 4-bit, none identical. Tier 1 ran Q4_K_XL, tier 2 UD-Q4_K_M at 15.77 GiB, tier 3 a 4-bit MLX affine build at group 64. This is a ladder, not a controlled curve, and I'd rather say so than dress it up.

The gaps, plainly:

Our own two rungs use **different harnesses**. Tier 1 came from a nine-prompt harness; tier 2 from `llama-bench` at `-r 3`. Our dataset methodology records the same 3090 running the same model at about 33 tok/s under one and about 42 under the other, purely from prompt shape. So even 128.08 against 37.23 carries a wide error bar, and the gap between them is not a clean 3.4x.

The repo doesn't state the **harness, context length, or prompt shape** behind 31-35 and 5.1-6.3. The README is upfront that its numbers are "a reference point, not a performance ceiling," which is the right posture, but it leaves no route to reproduce.

And nobody has run all three tiers **on one machine**. That's the measurement that would turn this into a curve.

If you have an M-series Mac, that's a run we can't make and you can. The dataset is CC BY 4.0 and takes firsthand numbers with the harness written down, negative results included.

The bottom line

On an 8 GB Mac, this makes a 26B model **run**. It does not make it pleasant. Five to six tokens a second is watchable rather than usable, and reporting that number honestly is to the author's credit. On a 24 GB M5 Pro at 31-35 tok/s it's something you'd actually use.

On a PC with a 12GB card you don't need it, but you do need 11.2 GB of VRAM free. If you haven't got that, you have nothing. That's the gap the SSD tier fills.

The memory hierarchy under local inference now runs VRAM, RAM, disk. On MoE models each step down costs far less throughput than the bandwidth gap suggests it should, and buys far more headroom. That's a property of the architecture, and it's why MoE keeps winning ground in local inference.

Get notified when we publish new guides.

Subscribe — free, no spam
