SPECULATIVE DECODING · HOMELAB

Speculating on a Strix Halo (part 2)

Measuring the fast engine

By Nate MacFadden · August 2026 · code private for now, happy to discuss

🤖 Human + AI (Claude) joint written 🤖

What the first measurement got wrong

Part 1 was all PyTorch. Here I switch to llama.cpp on Vulkan, which is what I run day to day, and the first thing I tried was the obvious experiment: start llama-server with the DFlash drafter attached, run a set of prompts, restart it without the drafter, run the same prompts, and compare tokens per second. The prompts were six I had Claude generate for benchmarking, so the absolute rates aren't workload numbers, but both configurations saw the same six, which is all a comparison needs.

NAME	PROMPT
code_merge	Write a Python function that merges two sorted lists into one sorted list. Explain it briefly.
code_class	Write a small Python class representing a 2D vector with add, subtract, and dot product methods.
prose_expl	Explain in a few paragraphs why memory bandwidth, not compute, usually limits single-stream LLM decoding.
prose_story	Write a short paragraph describing a thunderstorm arriving over a quiet coastal town.
factual	List the main differences between a mixture-of-experts transformer and a dense transformer.
stepwise	A train travels 120 km at 60 km/h, then 180 km at 90 km/h. Work through the total travel time step by step.

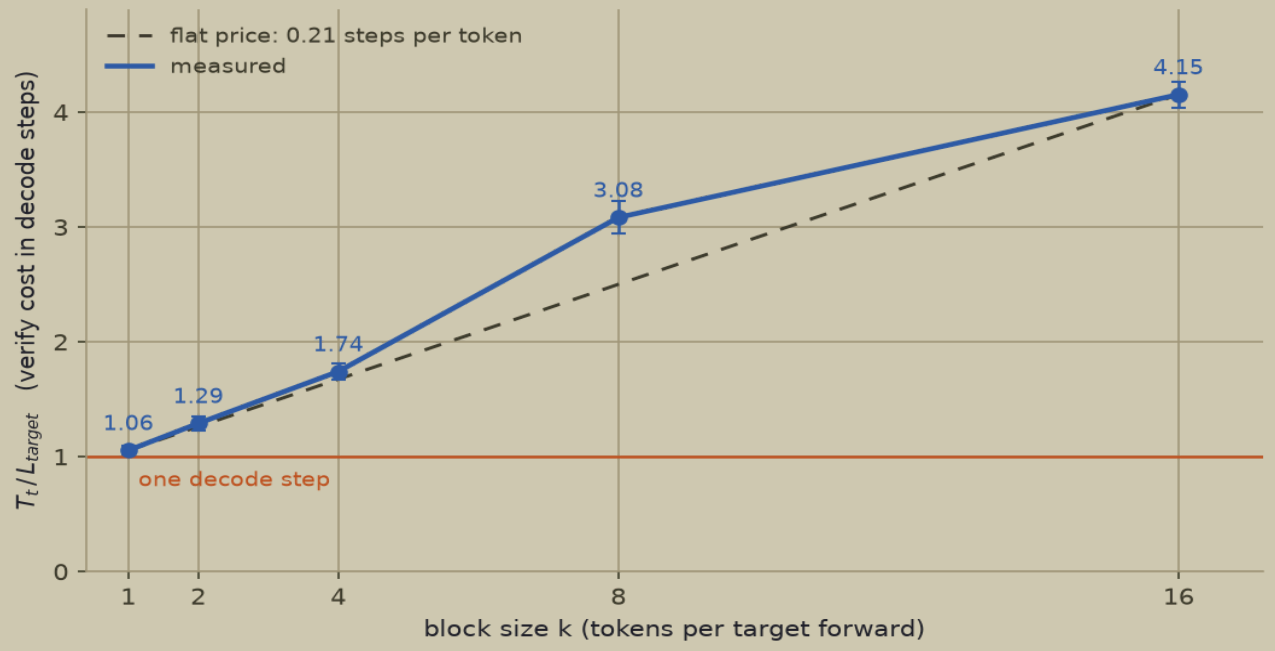
The rates were so broken that I don't even show it here... specifically,

1. **The outputs changed.** With speculation on, all six prompts produced different text than the same prompts without it, and 14 of 80 requests came back after about one token, the server emitting EOS immediately on a prompt that normally runs to 256. Tokens/s wasn't comparing like with like.
2. **Throughput drifted.** In every speculative condition it fell over a session, worst case 22.8 tok/s down to 13.5 tok/s, while the baseline held flat at 46.3 to 46.2 tok/s. Averaging over a session mixed the fast start with the slow end.
3. `τ` **was inflated.** My accounting assumed every cycle drafts the full 15 tokens, but the draft clips near a request's token budget. That inflated the `τ` I was computing by about 0.6% pooled over the sweep, 4.6% on the worst request.

So no speedup or `τ` from those server runs is usable. For a sense of how unusable: the same setup gave 0.36× pooled and 1.7× on a fresh server. The failure was valuable anyway: chasing why the outputs changed turned up two llama.cpp bugs, and that chase gets its own section at the end. But it also meant end-to-end timing wasn't going to answer the speedup question on this engine, so I measured the pieces first.

What a verify costs

I broke the speedup into pieces I could measure one at a time. The verify cost needs no drafter at all: a verify is just the target pushing a whole block through in one forward, and llama-bench times that directly, so none of what went wrong above can reach it. I ran it with the same build, model and flags as the server, 512 tokens of context, pushing k tokens through per forward. One token at a time is a decode step, 21 ms. The cost of k at once, in decode steps:



How many times longer the target takes over a k-token block than over a single token, measured with llama-bench at 512 tokens of context; error bars are ±1 standard deviation over its ten repetitions. The k=1 point sits at 1.06 decode steps rather than 1.00 because llama-bench's k=1 row is a one-token prefill (22.25 ms) while the decode step is its tg32 row (21.08 ms). The dashed line is what a flat per-token price would predict. Cost accumulates fastest in the middle of the block; the last eight tokens are the cheapest per token.

Part 1's speedup was $\eta = \tau \cdot L_{target}/(T_t + T_d)$. Divide through by L_{target} and it is $\tau/(T_t/L_{target} + T_d/L_{target})$, so speculation only wins when τ beats that denominator. My drafter fixes k at 16, and the curve above puts T_t/L_{target} at 4.2 decode steps there. So a cycle has to commit more than 4.2 tokens before speculation gains anything at all.

If you recall from part 1, that ratio previously was 1.6 decode steps. Somehow we regressed in ratio... how? Well the speedup empirically is asymmetric. The verify pass sped up from 205 ms to 88 ms (2.3× faster) compared to the decode step speeding up from 125 ms to 21 ms (6× faster). A couple things changed at once (Q8 quantization and the engine) so we can't pinpoint it, but we can say some things. Specifically, if we take an unrealistic limit in which the engine is so good that there was no time spent in compute (not actually possible), then all the time would be spent in memory traffic and we can compare to the memory's rating of 256 GB/s. The larger the deviation, the larger the fraction of time spent in compute (and hence the more inefficient the engine). For decoding in

PyTorch, we would find 5.8 GB trafficked in 125 ms, giving 47 GB/s, well under the rating, indicating that much of the time is spent in compute. Contrast this to decoding in llama.cpp (Q8), for which we traffic 3.1 GB (less than 5.8 GB due to Q8 quantization) in 21 ms, giving 147 GB/s. This is still far off from the memory rating since we do spend time in compute, but llama.cpp is approaching it. This matters because the decode step reads only 2.9B parameters compared to the 7.8B parameters in the verify, explaining some of the asymmetry.

So what's left? The speedup is τ divided by the cycle's cost in decode steps. Part 1's 2.7× was $\tau = 4.9$ against a verify costing 1.6 decode steps plus a drafter costing 0.2 decode steps. Here the verify alone costs 4.2 decode steps, so even with a free drafter the same τ gives 4.9/4.2 ≈ 1.2×. And 4.9 is optimistic here, for two reasons. It was measured on the bf16 PyTorch stack, and the drafter conditions on the target's hidden states, which now come from a Q8 model; whether it accepts as often on those I haven't tested. And the only τ I measured on this engine, 2.00, came from the broken server runs, so it can't be trusted either way, but if the true value is anywhere near it, a 16-token block loses outright: 2/4.2 ≈ 0.48×. That is the main result of part 2. The verify costs 4.2 decode steps here against 1.6 decode steps in part 1, and that alone takes 2.7× down to about break-even. What follows is about whether a different block shape gets it back.

Why draft depth is the lever

One more drafted token costs about 0.21 decode steps of verify, the end-to-end slope of the curve above. If that token is accepted it saves a whole decode step, since the target never has to generate it. So a drafted token earns its place if it's accepted more than about a fifth of the time. The slope isn't constant, though: going from a block of 4 to a block of 8 adds 0.34 ± 0.04 decode steps per token, while going from 8 to 16 adds only 0.13 ± 0.02 decode steps per token, well outside the run-to-run spread. I don't know why it bends there.

The deepest drafted token only commits if the fourteen before it were accepted too. Over 7000 cycles, 31% accepted nothing, 37% accepted exactly one token, and 3.5% took the whole block. Those runs include the requests that terminated early, so I also checked without them: 22%, 39%, and 5%. These counts come from the runs I said I wouldn't show. The rates from those runs are junk, and the counts are shakier than I'd like, since the stale-K/V bug below perturbs the very logits that decide acceptance. But that perturbation is last-bit noise, and the margin here is 3.5% against 13%, so I use them. Either way the deep end is nowhere near often enough. At the 16th position the block costs about 0.13 decode steps per extra token, so it needs ~13%, and it delivers 3.5%.

Draft shallower and the trade nets out ahead. If acceptance were geometric, the bf16 τ of 4.9 at depth 15 implies an acceptance probability of about 0.80 per position. A depth-4 draft is a 5-position block, which interpolates the verify curve at 2.1 decode steps, and at 0.80 it commits $\tau \approx 3.4$ tokens per cycle: 3.4/2.1 ≈ 1.6×, against 1.2× at depth 15. Or hold the depth and push acceptance toward 0.95 per position, again a probability: $\tau \approx 4.5$ at depth 4 for 4.5/2.1 ≈ 2.2×, and $\tau \approx 11$ at depth 15 for 11/4.2 ≈ 2.7×. Both τ figures come from that geometric model, not measurement, and the counts above say the model is wrong in shape: geometric at $\tau = 4.9$ predicts 20% of cycles accepting nothing and 16% exactly one, against the 31% and 37% I measured. Treat them as sketches. Both also assume a free drafter: I never measured T_d on this engine, and on a decode this cheap it is unlikely to stay the 0.2 decode steps it was in bf16.

Two bugs upstream

Checking whether speculation was even lossless here turned up two llama.cpp bugs, and I filed both.

The first is a recurrent-state rollback slot that is reported restored but never written[1]: a 68-line reproducer, bisected to the commit that added those per-draft-position state snapshots. It was triaged low priority: the snapshots are "only used for speculative decoding, where we never have to remove the anchor token," and the feature "is anyway marked experimental." My proposed fix[2] cost ~5% of decode to close something the only consumer of those snapshots cannot reach, and was declined: "~5% is ~5% too much for a bug that doesn't exist when using this feature."

The second is a Vulkan flash-attention issue[3] where stale K/V left in cache slots that were freed, which speculative rollback produces every cycle, still changes the logits. The root cause is below llama.cpp, in the driver or the hardware. On this machine's RADV stack (Mesa's AMD Vulkan driver) on RDNA 3.5 hardware, the cooperative-matrix multiply-add (the shader path llama.cpp picks on this GPU, `cm1` in the table below) returns different results depending on the sign of zero-valued inputs. A standalone 16x16 test reproduces that with no llama.cpp code involved: one column of -1.0, one row of small constants, everything else zero. Flip a row of those zeros to -0.0 and exact answers like -0.25 come back one bit off, while the same binaries on an NVIDIA card are bit-exact either way.

```
device: Radeon 8060S Graphics (RADV STRIX_HALO) | driver: radv Mesa 26.0.3-1ubuntu1 | a
computing, on the gpu, twice:
D[i,j] = sum over k of A[i,k] * B[k,j]
for 16x16 A, B with f16 elements.
A[i,12] = -1.0 and B[12,j] = (j+1)/16 are the only nonzeros, so every
D[i,j] = -(j+1)/16 exactly; the other 15 terms of each sum are zeros.
input check: the runs' B buffers differ in 16/256 elements, each 0x0000 (+0.0) -> 0x80
A is identical in both runs and is +0.0 in every column those elements multiply

  elem         gpu, +0 run  (bits)        gpu, -0 run  (bits)      |ulp|   cpu (double
  D[ 0, 0]     -0.06250000  0xbd800000    -0.06250001  0xbd800001      1    -0.0625000
  D[ 0, 1]     -0.12500000  0xbe000000    -0.12500001  0xbe000001      1    -0.1250000
  D[ 0, 2]     -0.18750000  0xbe400000    -0.18750001  0xbe400001      1    -0.1875000
  D[ 0, 3]     -0.25000000  0xbe800000    -0.25000003  0xbe800001      1    -0.2500000
                                    [trimmed: it prints 10 of the 256]

differing elements: 256 / 256   (max |gpu(+0) - cpu| = 0.000e+00)
signed-zero sensitivity reproduced
```

Whose bug is it?

Back to the second bug. Running the original llama.cpp reproducer, real model and real rollback rather than a synthetic matrix, agrees. Same model, same prompt, same rollback, on everything I could run it on:

DEVICE	DRIVER	ATTENTION PATH	ROLLBACK RESULT
Radeon 8060S (RDNA 3.5)	RADV	coopmat (cm1)	**leaks**
Radeon 8060S (RDNA 3.5)	RADV	scalar	exact
RTX 5060 Ti	NVIDIA	coopmat2	exact
RTX 5060 Ti	NVIDIA	coopmat (cm1), forced	exact
RX 6700 XT (RDNA 2)	RADV	scalar	exact
Intel Arrow Lake iGPU	Mesa	scalar	exact
lavapipe (software)	Mesa	coopmat (cm1)	exact
three x86 CPUs	-	CPU backend	exact

Forcing an NVIDIA card onto the same cm1 shader with `GGML_VK_DISABLE_COOPMAT2=1` gives exact results, so the shader source isn't the problem. The obvious objection is that I bisected this to a llama.cpp commit, so how can it be the driver's fault? That commit rewrote the Vulkan flash-attention shader so that masked-out columns now pass through the matrix instruction. The instruction had presumably always behaved this way; nothing had fed it those inputs before. My standalone reproducer triggers it through the f32-accumulator instruction. I could not isolate the exact f16 configuration the attention shader uses, but sign-of-zero is confirmed as the trigger on the attention op itself.

And RADV is allowed to do this. SPV_KHR_cooperative_matrix[4] says the order of the adds is implementation-dependent and leaves precision to Vulkan, and Vulkan doesn't specify it either. Vulkan also permits an implementation to ignore the sign of zero here. There is an opt-out flag, but it covers a fixed list of simpler operations and matrix multiply isn't on it. NVIDIA happens to be bit-exact on the same test, so the shader has been depending on something the spec doesn't guarantee. The fix is for the shader to zero its masked columns itself.

How this reaches speculation: rollback frees cache slots but doesn't clear them, so they still hold old K/V values, many of them negative. The mask sets those slots' attention weights to +0, and +0 times a negative number is -0, which is the input that triggers the difference. A few last-bit flips per attention op add up to logit shifts of up to about 8e-2 on a 4B model. I didn't measure the 35B model. That is large enough to flip a greedy token at a near-tie, but I never saw it happen.

Neither bug explains the degenerate replies that started all this. On a fresh server the speculative and plain runs agree for 26 tokens. The trace shows token 26 came from a cycle that accepted zero drafted tokens, so it is the target's own choice from an identical prefix, not a drafted token getting through. Prefill is fine. What's left is verify or rollback failing to restore state, the llama.cpp version of what I patched around in transformers. Why token 26 and not earlier, I don't know. This is still open.

References

1. ggml-org/llama.cpp issue #26695, *Eval bug: llama_memory_seq_rm returns true but doesn't restore the recurrent state when rolling back entire last decode.* Filed by me, August 2026. Maintainer triage, 2026-08-08: "The `n_rs_seq` stuff is only used for speculative decoding, where we never have to remove the anchor token i.e the first token. I consider this a low priority bug since `n_rs_seq` is anyway marked experimental." On the proposed fix: "~5% is ~5% too much for a bug that doesn't exist when using this feature. It would likely not be merged."

2. ggml-org/llama.cpp PR #25004, *recurrent : support equal splits for recurrent-state rollback.* The linked comment carries my one-slot alternative and the timing table: 16.06 ms/token at `n_rs_seq=15` on the PR base, 47.41 on the PR, 16.79 with the one-slot fix (Qwen3.5-4B Q4_K_M, Vulkan, Radeon 8060S).

3. ggml-org/llama.cpp issue #26744, *Eval bug: vulkan flash attention lets stale K/V in freed cells influence the output.* Filed by me, August 2026. "This bug does NOT occur if either `GGML_VK_DISABLE_COOPMAT=1` or the computation is done on the CPU."

4. Khronos. SPV_KHR_cooperative_matrix. "The order of the operations is implementation-dependent." "The internal precision of floating-point operations is defined by the client API."