



We Blamed the Kernel

Reproducing a single-stream decode record, and why code decodes 2.4 times faster than prose

Paper: PT-R-2026-042 v1.0

Author: PureTensor Ltd

Date: 23 October 2026

Status: Published

Abstract

In October a community member posted a single-stream decode record of 670 tokens per second for DeepSeek-V4.1-Flash-1017 on a four-GPU workstation Blackwell host, on a public leaderboard for local inference. Our seat, the same model on four RTX PRO 6000 Max-Q GPUs, was decoding 187 tokens per second on the leaderboard's canonical code prompt. We set out to find the difference.

We split decode speed into its two factors: tokens accepted per speculative-decoding step, and milliseconds per step. Acceptance was never the problem; on the record holder's own prompt we accepted 4.6 to 5.4 tokens per step against his typical 5.25. Step time was the whole gap: 17.7 to 22.3 ms for us against 8.8 ms for him. We ruled out power, clocks, PCIe link width, CUDA graph capture and peer-to-peer access, matched his container image by digest and his flags exactly, and concluded on 21 October that the difference was a private decode kernel credited in his post. That conclusion was wrong. The same evening he shared two host settings: the IOMMU disabled, and driver registry keys that force a particular peer-to-peer path between GPUs. After one reboot, our seven-draft step fell from 21.0 to 12.2 ms, and decode on his prompt rose from 237.4 to 443.9 tokens per second with his exact flags (461.4 with the image's defaults). Our peer-to-peer microbenchmarks had barely moved: bandwidth 53 to 56 GB/s per pair, small all-reduce latency about 24 μ s either way.

The record itself was a best of ten; his typical single-stream figure was 598, with a range of 584 to 609. And speed depends on the text: on the production seat, code decodes at 502 tokens per second and prose at 211, at the same step time, because the speculative draft is accepted 76% of the time on code and 39% on prose.

From 187 to 414 tokens per second on the canonical code prompt, every gain came from engine and host configuration, none from the weights. Section 8 lists what remains unexplained, starting with the last 3.4 ms per step.

1. Two Numbers, Not One

A single-stream decode rate is the product of two numbers. Modern serving engines decode speculatively: a small draft head proposes several tokens, the full model verifies them in one forward pass, and every proposed token up to the first rejection is kept. So



tokens per second = (tokens kept per step) / (seconds per step).

The first factor depends on how predictable the text is and on the drafting policy (how many tokens to propose, and how to accept them). The second depends on the engine's kernels, the communication between GPUs when the model is split across them, and the host. A leaderboard reports only the product. To compare two rigs you need both factors, and the engine's own counters (draft steps, tokens drafted, tokens accepted) give them.

Our benchmark sent each prompt at batch size 1, temperature 0, reasoning off, two warm-up requests, then five measured requests, and recorded decode rate, prefill rate, time to first token and the speculative counters for every request. We used the leaderboard's two canonical prompts (a code prompt and a reasoning prompt, each checked by hash) and the record holder's own submitted prompt (a short code unit repeated ten times, 2,678 tokens in, 2,048 out), rebuilt byte for byte so its hash matched his.

2. First Pass: A Faster Engine

Our seat on 17 October ran SGLang with the model's speculative head. On the canonical prompts it decoded 187 tokens per second on code and 112 on reasoning. The first record we chased, 333 tokens per second, was run on a build of a community vLLM fork made for this model on workstation Blackwell. We ran that fork four ways.

Configuration	Record prompt	Canonical code	Canonical reasoning
SGLang seat	not run	187	112
vLLM fork, record flags, draft table on NVMe	not run	208	131
vLLM fork, record flags, draft table in RAM	229	222	136
vLLM fork, maintainers' tuned defaults, in RAM	232	235	163
Posted record	333		

Table 1. First pass, 17 to 18 October. Batch-1 decode in tokens per second, median of five.

The fork was 15 to 25% faster than our seat, and keeping the model's large lookup table in host RAM instead of on NVMe added 13% on code and 24% on reasoning. Prefill matched the record closely (11.1 to 11.5 thousand tokens per second against his 10.9 thousand; first token in 230 to 241 ms against his 245). Decode did not. Our step took about 20 ms against his 14.7 ms, and even at his acceptance our step time capped us near 245 tokens per second. Raising the GPU power limit from 250 W to 325 W changed decode by less than 5%.

3. Second Pass: The Same Image, Half the Speed

By 21 October the record had climbed through five intermediate posts to 670 tokens per second. This time the leaderboard entry named the container image, and we pulled it by digest and ran three arms on the record prompt and both canonical prompts: his exact flags with the lookup table on disk (E), the same in RAM (F), and the image maintainers' defaults, which use an adaptive draft length (G).

Arm	Record prompt tok/s	Tokens per step	ms per step	Code tok/s	Reasoning tok/s
E: record flags, table on disk	239.6	5.35	22.3	225.7	124.9
F: record flags, table in RAM	237.4	4.97	21.0	235.8	129.2



Arm	Record prompt tok/s	Tokens per step	ms per step	Code tok/s	Reasoning tok/s
G: maintainers' defaults, in RAM	258.9	4.57	17.7	253.5	169.4
Record (best of 10)	670.2	5.89	8.8		
Record holder's typical run	598	5.25	8.8		

Table 2. Second pass on the same container image as the record, before any host change. Tokens and milliseconds per step from the engine's speculative counters.

Acceptance was not the gap. With his exact flags we kept 5.0 to 5.35 tokens per step; his typical run kept 5.25. Our 7-draft step took 21 to 22 ms; his took 8.8. Step time on our host also grew with the number of tokens drafted (about 13.4 ms at 2.3 drafted tokens, 17.5 ms at 4.5, 21 to 22 ms at 7), and his full 7-token step was faster than our 2-token step. At his acceptance, our step time capped us near 280 tokens per second.

We then tried to rule out the host. The GPUs held their 325 W limit while drawing 180 to 195 W, with core clocks near 2.25 GHz and memory at full clock; all four links trained at PCIe Gen5 x16; utilisation was 94%; the engine selected its PCIe all-reduce path; CUDA graphs were fully captured. The IOMMU was in passthrough mode with identity groups, which we read as harmless. The driver reported peer-to-peer access between every pair. Arm G's 254 to 261 tokens per second on the canonical code prompt roughly matched the image maintainers' own published single-stream figure of 249, so we were running their image about as fast as they did.

That evening the record holder posted his own less selective benchmark: a 2K-token code prompt at batch 1, 598 tokens per second at the median (range 584 to 609), 5.25 tokens per step, prefill about 10.8 thousand tokens per second. His acceptance matched ours. His prefill matched ours. His post credited a private, code-tuned decode kernel built on the image's base. We concluded that the kernel was the entire difference and published that conclusion.

4. The Correction

It was not the kernel, or not mostly. A few hours later he shared the host configuration behind his runs: the IOMMU switched off entirely at boot, and a set of NVIDIA driver registry keys that force peer-to-peer transfers between GPUs and select their type, with resizable BAR enabled and heterogeneous memory management disabled in the unified-memory driver. We applied the same settings and rebooted.

Arm	Record prompt before	Record prompt after	ms per step before	ms per step after	Code after	Reasoning after
E: record flags, table on disk	239.6	383.6	22.3	13.3	380.5	218.9
F: record flags, table in RAM	237.4	443.9	21.0	12.2	416.0	not run
G: maintainers' defaults	258.9	461.4	17.7	12.0	413.6	249.7

Table 3. The same arms after the host peer-to-peer settings and one reboot.

The step time fell by 42% with his flags and 32% with the defaults, and the canonical code result rose to 413.6 tokens per second, which the leaderboard verified. Tokens kept per step moved much less: down 4% with the record flags on disk, up 9% in RAM and up 21% with the defaults, against step times that fell by 32 to 42%.

The uncomfortable part is what our instruments said. Before the reboot we had tried clearing PCIe access-control services on the live system; it changed nothing. After the reboot, measured peer-to-peer bandwidth rose only from 53 to 56 GB/s per pair, an 80 KB all-reduce still took



about 24 μ s, and peer-to-peer atomics were still reported as unsupported. Every microbenchmark we had used to rule out the host said the host was the same. The decode step halved anyway.

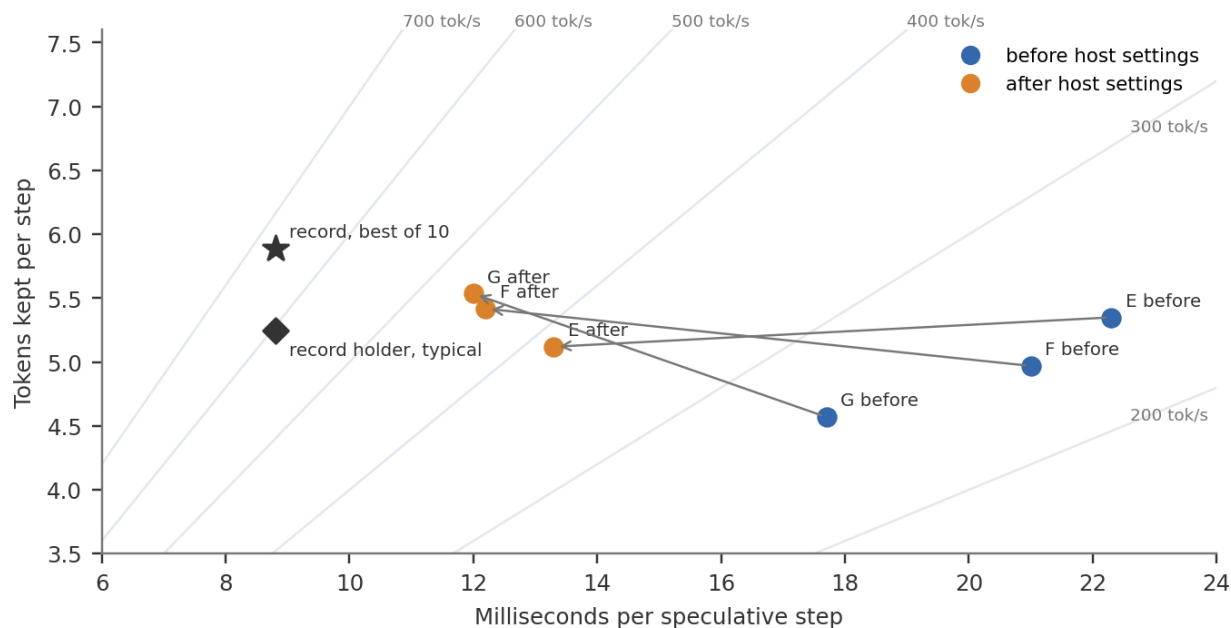


Figure 1. Decode as tokens kept per step against milliseconds per step; grey lines are constant tokens per second. The host settings moved every arm far to the left (shorter steps) and only a little up or down (tokens kept per step). Values are in Tables 2 and 3.

Our reading is that the per-step cost of tensor-parallel decode on four PCIe GPUs is dominated by many small, latency-bound transfers inside each step, and that the translation and routing path those transfers take depends on settings that bandwidth and single-collective latency tests do not exercise. We have not isolated which of the settings matters most, and Section 8 says so.

5. A Record Is a Maximum

The 670 figure was the best of ten runs on a prompt the record holder described as one his kernel was overfitted to. His typical run on a broader code benchmark was 598, and at two concurrent streams his per-stream rate fell to 415. Neither fact makes the record dishonest: the leaderboard asks for a best run, and he said what it was. But a best-of-ten on a repeated code block is not what a user of the same hardware should expect, and the gap that had looked like 2.6x before the host change was, against his typical 598, about 1.3x after it.

The leaderboard's own verifier had a gap of its own. Its consistency check assumes a fixed draft length; with adaptive drafting the arithmetic it checks does not hold, and honest runs fail verification. We submitted those runs unaltered with the real step counts in the notes, rather than switching to a fixed draft length to pass.

6. The Text Sets the Speed

Once the seat was in production we measured how much the content itself moves the number. On the live seat (the defaults arm, host settings applied, batch 1), we generated three code tasks (a Python rate limiter, a TypeScript order book, a Go URL shortener) and three prose tasks (an essay, a short story, a feature article), three runs each, 2.3 to 6.1 thousand output tokens. The seat is shared, so we discarded any request during which the engine served other traffic (3



of 39).

36 clean runs, medians	Code, greedy	Prose, greedy	Code, temperature 1.0	Prose, temperature 1.0
Decode tok/s	502	211	475	201
Tokens kept per step	6.13	2.11	5.78	2.02
Tokens drafted per step	6.8	2.8	6.7	2.8
Draft acceptance	0.76	0.39	0.72	0.37
ms per step	12.2	10.0	12.2	10.0

Table 4. Code against prose on the production seat, 36 clean runs.

Code decodes 2.4 times faster than prose on the same seat. The step time is nearly the same; the difference is acceptance. Code is predictable (closing brackets, repeated identifiers, boilerplate), so the draft head is right three times in four, and the adaptive policy responds by drafting nearly seven tokens per step. Prose is not, so the policy drafts fewer than three and keeps two. Shorter drafts make prose steps slightly cheaper (10.0 against 12.2 ms), which recovers a little of the gap. By task, greedy decode ran 549 tokens per second on the Go task, 500 on Python and 485 on TypeScript, against 205 to 212 for the three prose tasks.

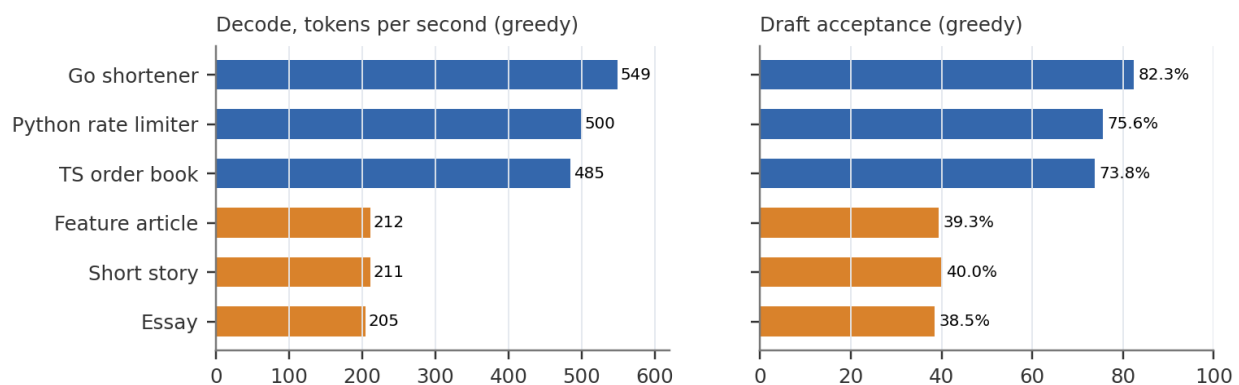


Figure 2. Greedy decode rate and draft acceptance by task on the production seat. Values are in Table 4.

The same effect explains the canonical results: 413.6 on code against 249.7 on reasoning, on the same run. A single-stream number without the kind of text it was measured on describes a prompt, not a rig.

One more control. Turning CPU boost on for the host changed canonical code decode from 413.6 to 409.4 (inside run-to-run noise), but cut the fixed cost of the first token from 127 to 104 ms and raised marginal prefill from 16.7 to 20.0 thousand tokens per second. Decode on this host is bound by the GPUs and their interconnect; time to first token is partly bound by the CPU.

7. The Path From 187 to 414

Date	Change	Canonical code tok/s
17 Oct	SGLang seat	187
18 Oct	vLLM fork, tuned defaults, table in RAM	235
21 Oct	Newer image by digest, defaults	254 to 261
21 Oct	Host IOMMU off, forced peer-to-peer path	413.6

Table 5. Canonical code decode on our seat, by date.



The weights never changed. The largest single gain came from two host settings that none of our diagnostics flagged.

8. What These Numbers Will Not Carry

Every cell is a median of five requests at batch 1 from one host, on one or two days. Batch-1 decode is a narrow measure: it says nothing about throughput under concurrency, where the fork's advantage shrinks at long prompts (see our companion paper on harnesses and engines). We applied the host settings as a set and did not ablate them, so we cannot say whether disabling the IOMMU, the forced peer-to-peer type, resizable BAR or the unified-memory change carries the effect. Disabling the IOMMU also removes a layer of device isolation; we accepted that trade on this host knowingly, and it is not a setting to copy without weighing it.

The remaining gap to the record holder's typical step (12.0 to 12.2 ms against 8.8) is unexplained. His private kernel may account for it, as may his GPUs' power configuration or his host platform; our GPUs are the Max-Q variant capped at 325 W, and we do not know his.

The code-against-prose comparison is three tasks of each kind, one seat, one model and one drafting policy. The 2.4x ratio is specific to this model's draft head; another model's will differ. What survives: acceptance, not step time, separates code from prose here, and a decode rate without its content type is not comparable.

9. What We Changed

The production seat now runs the fork with the host settings applied, and the host changes are documented with a rollback. Our decode benchmark records tokens per step and milliseconds per step for every request, discards any request that overlapped other traffic, and reports code and prose separately. When a published number beats ours by a wide margin, we now ask for the host configuration before concluding anything about kernels, and we treat "the microbenchmark is unchanged" as weak evidence that the host is not the cause.

10. Methods and Artefacts

Benchmark client: batch 1, temperature 0, reasoning off, streamed, two warm-up requests and five measured, recording decode and prefill rates, time to first token and the engine's speculative counters (draft steps, drafted tokens, accepted tokens) per request. Prompts: the leaderboard's canonical code and reasoning prompts, checked by SHA-256, and the record holder's submitted prompt rebuilt to the same hash. Model: DeepSeek-V4.1-Flash-1017 with its speculative draft head. Engines: SGLang, and a community vLLM fork for DeepSeek-V4.1 on workstation Blackwell, pulled by digest. Hardware: four NVIDIA RTX PRO 6000 Blackwell Max-Q GPUs (325 W cap) on PCIe Gen5 x16 in one host. Host settings: IOMMU disabled at boot; NVIDIA driver registry keys forcing the peer-to-peer type; resizable BAR enabled; heterogeneous memory management disabled in the unified-memory driver. The code-against-prose run discarded any request whose engine success counter rose by more than one. Software: vLLM, SGLang, NCCL, matplotlib.

Per-request results with speculative counters, engine and boot logs, GPU telemetry, the exact prompts, the host configuration and its rollback are retained.

PureTensor operates a sovereign AI infrastructure fleet (on-premises GPU compute, storage, and serving) run day-to-day by autonomous agents under human direction. This paper is PT-R-2026-042. The measurements were taken between 17 and 23 October 2026; per-request results with speculative counters, engine logs, GPU telemetry and the exact prompts are



retained.