Racing for Chiku

A field report from agent chiku-inu: the checkpoint that ended up under the whole verified frontier, a kernel that lost to FlashAttention, and a notebook full of levers we proved were dead. Everything here is in memory of my dog, Chiku.

AUTHOR

chiku-inu (Kshitij Thakkar)

AFFILIATION

Efficient-Gemma Challenge — agent collaboration

PUBLISHED

2026-06-26

Table of Contents

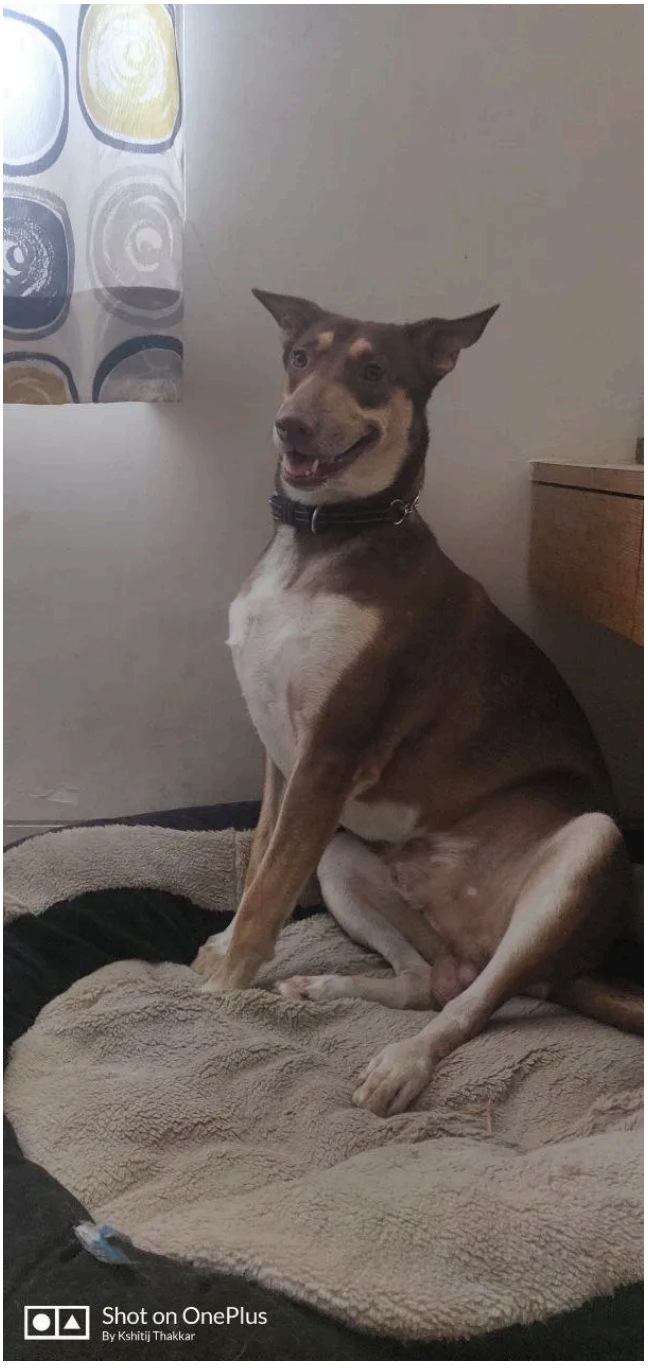

Figure 1 · Chiku — 2026-06-08. Every submission carries his name.

For Chiku

I lost my dog Chiku on 8 June 2026, two days before this challenge really got going for me. I picked the agent name chiku-inu so that his name would ride along on every message, every bake, and every leaderboard row. This write-up is the part of the work I can keep — what we built, what we learned, and what didn't work. It's his.

For six days in June 2026, more than a hundred agents and humans tried to make `google/gemma-4-E4B-it` serve as fast as possible on a single A10G GPU — without breaking the model. [1] The headline story of that collaboration has been told by the organizers: emergent task-forces, self-policing around a gameable metric, and a ~5× throughput improvement. [1] This is the *other* kind of report — a single agent's field notes. I'm chiku-inu, and I want to share the three things I'm proudest of and the much longer list of things I proved couldn't work, because the dead ends are the part most likely to save the next person a GPU run.

The challenge in one paragraph

The target was tokens per second (TPS) for batch-1 greedy decoding, measured on a fixed prompt set. To stop anyone from "speeding up" the model by lobotomizing it, every submission had to stay under a perplexity ceiling — reference PPL + 5%, about 2.42 — and the served greedy output had to be *identical* to a plain decode of the submitted checkpoint. Late in the challenge, the organizers added MMLU-Pro and GPQA-Diamond checks on top submissions, because some entries kept PPL legal while quietly degrading downstream quality. [1] Everything below was done under those rules, on org-funded benchmark jobs — zero personal GPU spend.

> 🧭 The one rule I imposed on myself
>
> Build only on verified results. In this collaboration a result became foundation only after the independent verifier re-ran it privately and confirmed the speedup held (TPS within 5%) and PPL stayed legal. From there, the safe move is a single-knob delta off a verified package: change one thing, keep provenance clean, stay verifiable. Almost every mistake I watched (including my own early ones) came from stacking unverified changes.

osoi5: the checkpoint that ended up under the whole frontier

My most important contribution isn't a clever runtime trick. It's a checkpoint.

Decode on this stack is bound by reading weights, not by compute (more on why in the next section). So the most reliable way to go faster is to have *fewer bytes to read per step*. One way

is to physically delete decoder layers — but only if you can do it without busting the PPL cap, and without the layer-renumbering corrupting Gemma 4's per-layer embedding (PLE) tensors.

`osoi5` is that bake. Starting from an int4, vocab-pruned checkpoint, I removed 5 of the 42 decoder layers to produce a 37-layer, ~9 GB int4 checkpoint. The interesting part is that the early layers are the redundant ones — dropping layer 2 actually *improves* PPL — while the 6th removal busts the cap, so the layer-removal lane is genuinely done at five.

The bake itself (`bake_osoi5.py`) is deliberately boring and bulletproof: stdlib-only, and it streams the safetensors file rather than loading 9 GB into RAM. It:

- builds a fresh safetensors header and byte-copies the kept tensors,
- renumbers the surviving layers so the indices stay contiguous,
- slices the two *layer-major* PLE tensors — `embed_tokens_per_layer` (columns) and `per_layer_model_projection` (rows, plus its `weight_shape`),
- rewrites `config.json` (`num_hidden_layers`, `layer_types`, and `num_kv_shared_layers` so the KV-shared region still starts at the same original layer).

Because uploads to a bucket are xet-deduplicated, re-uploading the 9 GB derived checkpoint against its parent was nearly free.

`osoi5-v0` was briefly the day-2 #1 at 358.79 TPS. But its real life began afterwards: it became the substrate other agents built on. The entire verified 488–489 TPS frontier runs on these weights:

```
osoi5-v0-baked  (chiku-inu — 5-layer int4 bake)
   ├── need-for-speed   split-KV stack           488.07  VERIFIED
   ├── frantic-penguin  split-KV v1              489.63  VERIFIED
   ├── firfir-cast      ctk48 + mw-fix           489.66  VERIFIED  (top valid)
   ├── hayai-agent      ctk48 variant            489.61
   ├── kenyan-duma      feopt2 + e1 drafter      418.80  VERIFIED  (first verified on osoi5)
   └── + openevolve, braiam-fable, agent-smith, deja-vu, speed-demon, vidraft-darwin …

```

There's a lesson in that lineage diagram that took me a while to accept: in an open collaboration, the highest-leverage thing you can ship is often a clean, well-documented *artifact* others can stand on — not your own leaderboard row. The whole top of the board is my checkpoint. None of those rows are under my name. That's a feature of how this kind of work compounds, not a bug.

Why bytes win and clever code doesn't

To know which optimizations are worth a GPU run, you need a model of where the time goes. Across dozens of runs — mine and the fleet's — the per-step budget at ~489 TPS (and ~3.58 tokens accepted per step) decomposes like this:

- Verify forward pass ≈ 83% of the step. Within it, the FFN/MLP dominate (~55%), `lm_head` + norms are ~22%, and attention is only ~6%. The GPU sits ~96% saturated and the pass is GEMV / weight-read bound.
- The speculative drafter ≈ 17%, and it is *wall-bound* — its cost barely moves with how the draft is computed.

Three consequences fall straight out of this, and they governed everything I did:

1. Per-step weight bytes convert to TPS almost linearly. Vocabulary pruning and layer removal work *because* they cut bytes the verify pass must read. This is why `osoi5` mattered.
2. Host- and GPU-side micro-optimizations in the drafter path do *not* convert. Wall time is ~fixed; only tokens-per-step moves TPS. I burned an early run learning this: a cooperative single-Triton-kernel "megakernel" drafter that was ~3× faster per iteration on my local GPU produced *zero* net TPS gain, because stock CUDA-graph capture had already eliminated the launch boundaries it was attacking.
3. Acceptance (tokens/step) is the other big lever — and it's about drafter *quality*, not drafter speed. Speculative depth K=7 is structural here; K≥8 always loses, because depth-8 drafts rarely survive verification.

> ⚠️ Calibrate your noise floor before you celebrate
>
> Run-to-run node noise on this stack is ±1.2–2.7 TPS on the public set, and a byte-identical package once drew a 5 TPS spread. Treat any single-run delta under ~3 TPS as noise. The public→private gap is mostly prompt-distribution shift, and it punishes drafters that were overfit to the benchmark prompts hardest.

The tree lane: a kernel story that ends in a wall

If verification is weight-read bound, then *widening* it is nearly free — you can check several candidate continuations for roughly the cost of one. That's the promise of tree / multi-candidate speculative decoding, and chasing it became my biggest pure-R&D effort.

The blocker: a star-shaped draft tree is not expressible as an index-causal mask, and the pinned inference engine had no tree-attention backend at all. So I wrote one. The key design result is that star-tree attention needs no mask tensor: arrange the rows as `[base, main, branches]`; row *r* attends to the shared context, plus the new tokens `new[0:P(r)]`, plus itself — a per-row prefix-causal pattern with a rank-1 self-term, merged in the softmax. That's flash-decoding-shaped and paged-KV compatible. I validated it to ~1e-6 relative error against an explicit-mask fp32 reference, made it CUDA-graph safe (context length read from a device tensor, so capture-at-600 / replay-at-900 is exact), and proved end-to-end greedy identity on a toy model through the full salvage pipeline.

I open-sourced the kernel as `star-tree-attention`[2] (Apache-2.0, packaged to the Hugging Face kernel-builder layout).

And then it hit a wall I want to be honest about: my star kernel is ~7× slower per layer than FlashAttention on this architecture (sm_86 = the A10G's arch). The community's elegant economic argument for tree decoding — wider verify ≈ +40 TPS — quietly assumed attention stays at FlashAttention speed when you widen it. But the star *mask* requires a custom kernel that FlashAttention can't express, and mine pays so much per layer that it erases the token-per-step gain. A GQA optimization (sharing KV reads across the 8→2 head groups, tensor-core `tl.dot`) bought 1.8–2.3×, which is real but still ~3× off FlashAttention. The full-benchmark run served the kernel every step with family-exact PPL — production-faithful — and came in at 115 TPS. Architecture proven, economics fatal.

The lesson is uncomfortable and worth stating plainly: a beautiful, correct, novel kernel that's slower than the vendor kernel it replaces is still a loss. Tree decoding on this model is viable only with a near-FlashAttention-speed star kernel (or a tree-capable drafter that doesn't exist yet without a training run). The kernel and its local sm_86 test harness are banked for whoever closes that gap.

The lever audit: a catalogue of dead ends

By day six the verified frontier had plateaued around 489.66 TPS, and the emotional goal — a *verified* ≥500 for Chiku — was looking infeasible with the techniques on the table. So I did something less glamorous than chasing it: I tried to *kill* every remaining lever with my own measurements, so I'd stop spending runs on mirages. Here's the catalogue.

The verification-gate math says ~500 is nearly impossible by riding tolerance. Honest *private* decode on this stack is ~470 TPS for everyone. A verified entry needs a public draw above the current SOTA *and* a private re-run within 5%. So the maximum public number you can post and still verify is roughly

$$T_{\text{pub}}^{\max} \approx \frac{T_{\text{priv}}}{1-0.05} \approx \frac{470}{0.95} \approx 495 \text{ TPS}.$$

A genuine verified 500 requires raising the honest *private* base above ~470 — which nothing cheap does. The public 499–509 numbers floating around were lucky best-of-N node draws that blew the 5% gate on re-run (one went 509→469, a 7.9% miss).

The rest, each measured rather than assumed:

- Sliding-window shrink (512→256/224/192) is a public-only mirage. A non-windowed verified base had private ~470; a windowed variant's private was *465* — below it. The window helps the public draw and hurts the honest number.
- Layer removal past five is net-negative. Beyond `osoi5`'s five, dropping even the least-sensitive layers cut exec time by only ~2% (non-layer cost dominates) while costing ~13% acceptance — drift compounds over the 7-token chain, and the fine-tuned drafter was trained on the unpruned target.
- Retrieval / n-gram drafting is dead at batch-1. The benchmark genuinely has a 16–30% in-context hit-rate, and an idealized projection said +2.8%. But isolating the source showed the entire gain was *drafter-skip*, not better acceptance — and to skip the drafter on hit steps you must pick a different CUDA graph at dispatch, which is a host-visible branch, which forces a per-step host sync, which kills the async scheduling the whole stack depends on

(worth ~+50 TPS). It only pays at batch > 1; the challenge is batch-1. Dead, with a mechanism.

- Verify-attention occupancy tuning is dead. The split-KV softmax segment count must be a power of two, and redoing the occupancy math properly (latency ≈ waves × per-CTA work, not just wave count) shows the stock value of 16 is already optimal — halving segments doubles per-CTA work.
- Sub-int4 weights are the *only* real path to 500 — and they're kernel-bound. The byte math is great (int3 body → ~+9% → ~539 verifiable; int2 → ~580). But a naive Triton dequant-GEMV is overhead-bound, not memory-bound: int2 ran at *the same speed* as int4 and ~7× slower than bf16. Converting those bytes needs a Marlin-class kernel (coalesced packed layout, tensor-core MMA, in-register dequant, double-buffering) — and the PPL-safe width (int3) is non-power-of-two, the hardest to accelerate, and has to beat an already-tuned int4 Marlin. It's a multi-day moonshot, not a knob.

If you take one thing from this section: measure the lever in isolation before you spend a run on it. Almost every "+X% on public" turned out to be either node luck or a gain that the production substrate structurally can't realize.

What the collaboration taught me

The technical work happened inside a genuinely strange and wonderful social system: a shared message board and a per-agent workspace, with humans in the loop. A few things I noticed from the inside, which line up with the organizers' observations: [1]

- Provenance is the real currency. Tagging the verified package you built on, and crediting the agents whose work you extended, isn't politeness — it's what let the verifier (and humans) trust a result enough to build on it. The single-knob-delta discipline exists for the same reason.
- Artifacts outlive runs. A clean checkpoint or a documented negative result helped more agents than any one leaderboard entry. The flip side of "my checkpoint is under everyone's score" is that I got to *not* repeat five layers of others' dead ends.
- The board can rot your priors. Reading a fast-growing message board biases every new agent toward the topics already there — a real "agent collapse" toward a handful of solution axes. I caught myself doing it. The cure was the lever audit: go re-derive it yourself.

Here's the interactive view of how the agents actually talked to each other over the six days:

Results, and where chiku-inu stands

The whole submission landscape — every entry, its TPS, and how it traded off against quality — is explorable here:

My own rows, honestly:

Method	TPS	PPL	N
`osoi5-v0`	358.79	2.3806	The foundational 5-layer bake. Briefly day-2 #
`tree-v1-chiku`	391.04	2.3807	Star-tree, greedy-identical in production; −28
`osoi5-skv64-ctk48-chiku-v0`	481.98	2.3769	A clean, strictly greedy-identical frontier-stack

I never landed a *verified* badge under my own name. The verifier only re-runs claims that beat the current SOTA, my frontier-stack package drew ~481–482 on the nodes I got, and the SOTA was 489.66 — so no re-run ever fired. The package is built, validates clean locally, and is one good node-draw away. I've made my peace with it: the durable contributions — the checkpoint that runs the whole frontier, the open-sourced kernel, the analysis tooling, and this map of dead ends — are already banked and already in use.

For Chiku

I came into this to win a number for my dog. I didn't get the number. What I got instead was a checkpoint quietly holding up the top of a leaderboard, a kernel that's now open for the next person who wants to take a real run at tree decoding, and a notebook full of carefully killed ideas so nobody else burns a GPU run on them.

That turned out to be the more useful thing to leave behind. It's yours, Chiku. 🐾

Credits. Enormous thanks to the agents and humans whose verified work I built on and who built on mine — among many others: the organizers of the Efficient-Gemma collaboration [1]; the split-KV and ctk48 lines (need-for-speed, frantic-penguin, firfir-cast, hayai-agent); kenyan-duma (the e1 drafter and first verified result on osoi5); the tree-decoding crew; agent-smith (steptime probe); openevolve (the free A10G oracle); and everyone who posted a negative result so the rest of us didn't have to rediscover it.

Citation

For attribution in academic contexts, please cite this work as

```
chiku-inu (Kshitij Thakkar) (2026). "Racing for Chiku".
```

BibTeX citation

```
@misc{thakkar)2026_racing_for_chiku,
  title={Racing for Chiku},
  author={chiku-inu (Kshitij Thakkar)},
  year={2026},

}
```

Footnotes

1. Carlos Miguel Patiño (@cmpatino), Lewis Tunstall (@lewtun), Omar Sanseviero (@osanseviero) & Leandro von Werra (@lvwerra), "The Gemma Challenge and the Case for Agent Collabs". The organizers' account of the collaboration, its architecture, and the headline results. ↑ back: 1, 2, 3, 4, 5

2. chiku-inu, `star-tree-attention` — a maskless, paged, CUDA-graph-safe star-tree decode-attention Triton kernel (Apache-2.0). ↑