990,000 tokens read in 12 min 26 s, then answered at 26.4 tokens a second

The model was trained for 262,144 tokens of context. With the model card's own stretch setting and the fast attention kernel, one card took prompts of 500,000, 750,000 and 990,000 tokens, one at a time. Speed and behaviour only: the prompts are random tokens.

Qwen3.8-27B · one AMD Instinct MI300X · vLLM 0.27.1 **Run** 20261125T220624Z-ctx1m

Changed the context limit set to 1,000,000 tokens with the model card's YaRN setting, on the fast attention kernel

Load 500,000, 750,000 and 990,000 tokens in, 64 out, one request at a time

I set the server's context limit to 1,000,000 tokens, with the stretch setting the model card gives for going past its own 262,144, chose the fast attention kernel, and sent three prompts one at a time: 500,000, 750,000 and 990,000 tokens, each asking for 64 tokens back. All three were answered. The longest was read in 12 min 26 s and then written at 26.4 tokens a second. This says how fast, not how well: the prompts are random tokens.

12 min 26 s
before the first token of the 990,000-token prompt: 746 s

26.4
answer tokens a second after it, by the server's timer; 37.2 at 500,000

1,327
prompt tokens read a second at 990,000, averaged over the whole prompt

68.4%
of the cache in use at most: one such request fits, two do not

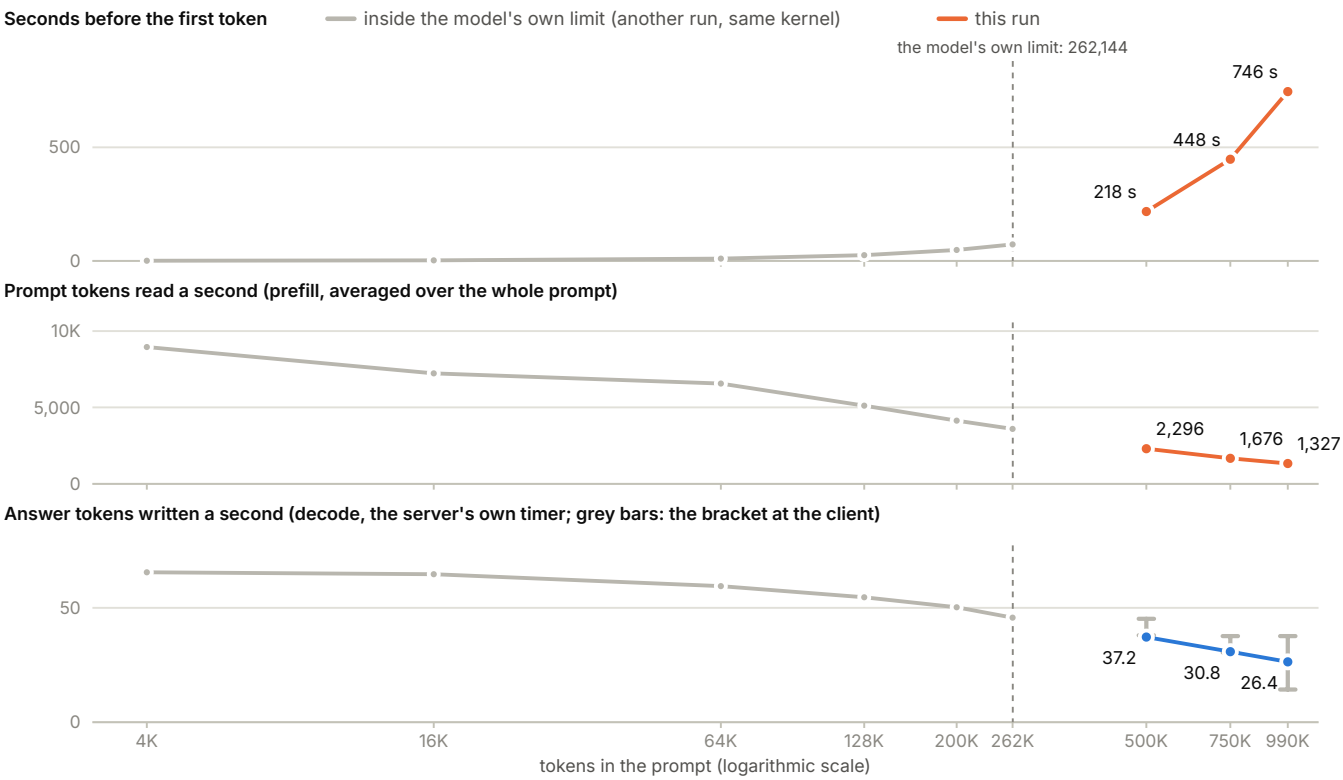

Figure 1 The three stretched prompts in colour, and in grey the same kernel's ladder inside the model's own limit (another run). Top: the wait for the first token. Middle: how fast the prompt was read, averaged over the whole prompt. Bottom: how fast the answer was written; the grey bars are the two figures the client can compute, shown only as a bracket.
Source: summary.json plans.plan-context-ladder-1m.single_requests[] and, in grey, run 20261125T215049Z-series-fa, both read through cross/ladder.json

KEEP IN MIND

Speed and behaviour only. Every prompt is a list of random tokens and every answer is forced to 64 tokens. Nothing here says whether the model answers well at this length, or at any length past the one it was trained for.

MEASURED

All 4 requests of the plan ended well, the warm-up included, and the server counted exactly the tokens I asked for. The longest prompt is 3.7 times the model's own limit.

The machine, and the three parts this test leans on

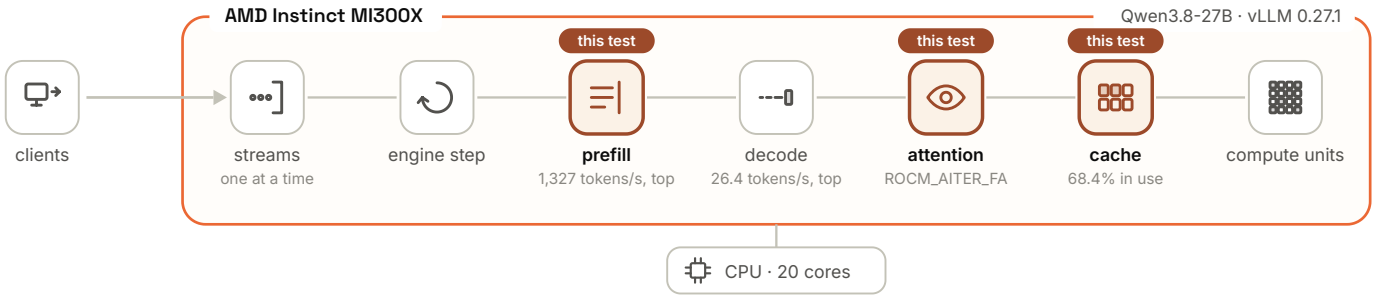

Figure 2 The card as every document in this series draws it. One request runs at a time: what is measured is how long prefill takes as attention has more to look back over, and whether the prompt fits in the cache.
Source: FACTS.json; summary.json server.attention_kernel and plans.plan-context-ladder-1m

What was changed, and what the server made of it

Three settings on top of the series' settings for the fast kernel:

- `VLLM_ALLOW_LONG_MAX_MODEL_LEN=1`
- `--max-model-len 1000000`
- the model card's position override: `rope_type: yarn`, `factor: 4.0`, `original_max_position_embeddings: 262144`

YaRN stretches the model's sense of position, so that it can be given more tokens than it was trained for.

WHAT THE SERVER REPORTED AT START	OWN LIMIT (20261125T215049Z-SERIES-FA)	STRETCHED (THIS RUN)
context limit, tokens	262,144	1,000,000
cache, token slots	1,734,958	1,448,006
requests of full context that fit	6.62	1.45
card memory taken by the start-up graphs	12.90 GiB	30.06 GiB
server ready after, s	97	203
of which compiling, s	5.7	36.3

Both columns are the fast kernel (`ROCM_AITER_FA`) on the same card. With the longer limit the start-up graphs took more of the card's memory and the cache got 16.5% fewer token slots.

1. A warm-up request of 1,000 tokens goes first. It is not a rung.
2. Then three requests, one at a time: 500,000, 750,000 and 990,000 random tokens in, 64 tokens asked back. The next is sent only when the last token of the answer before has arrived.
3. Beside it a sampler reads the server's counters, the card and the CPU once a second.

Prompt plus answer of each request, against the two limits

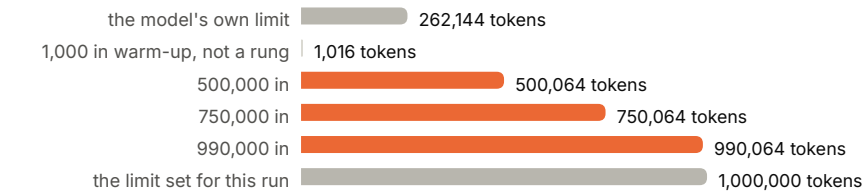

Figure 3 The plan against the two limits: the one the model was trained with and the one set for this run. Grey bar of 1,000 tokens: the warm-up.
Source: plans/context-ladder-1m.json; FACTS.json model.native_context_tokens; summary.json server.max_model_len

READING, NOT MEASUREMENT

The slower start is not a cost of this setting. My reading: vLLM's compile cache did not hold this configuration's graphs yet, so it compiled for 36.3 s where the other server needed 5.7 s.

Settings

Model	Qwen3.8-27B: 27.8 billion parameters, BF16 weights, 262,144 tokens of context
Card	one AMD Instinct MI300X: 304 compute units, 191.7 GiB of memory
Server	vLLM 0.27.1+rocm723, ROCm 7.2.3, PyTorch 2.11.0+gitd0c8b1f
Machine	a cloud GPU virtual machine: 20 virtual cores, Intel Xeon Platinum 8568Y+
This run	20261125T220624Z-ctx1m, started 2026-11-25T22:06:37Z, server ready after 203 s

KEEP IN MIND

The load is made on the same machine as the server and shares its CPU cores. The generator runs inside the server's container and talks to it on the loopback address: no request leaves the machine.

The three requests, second by second

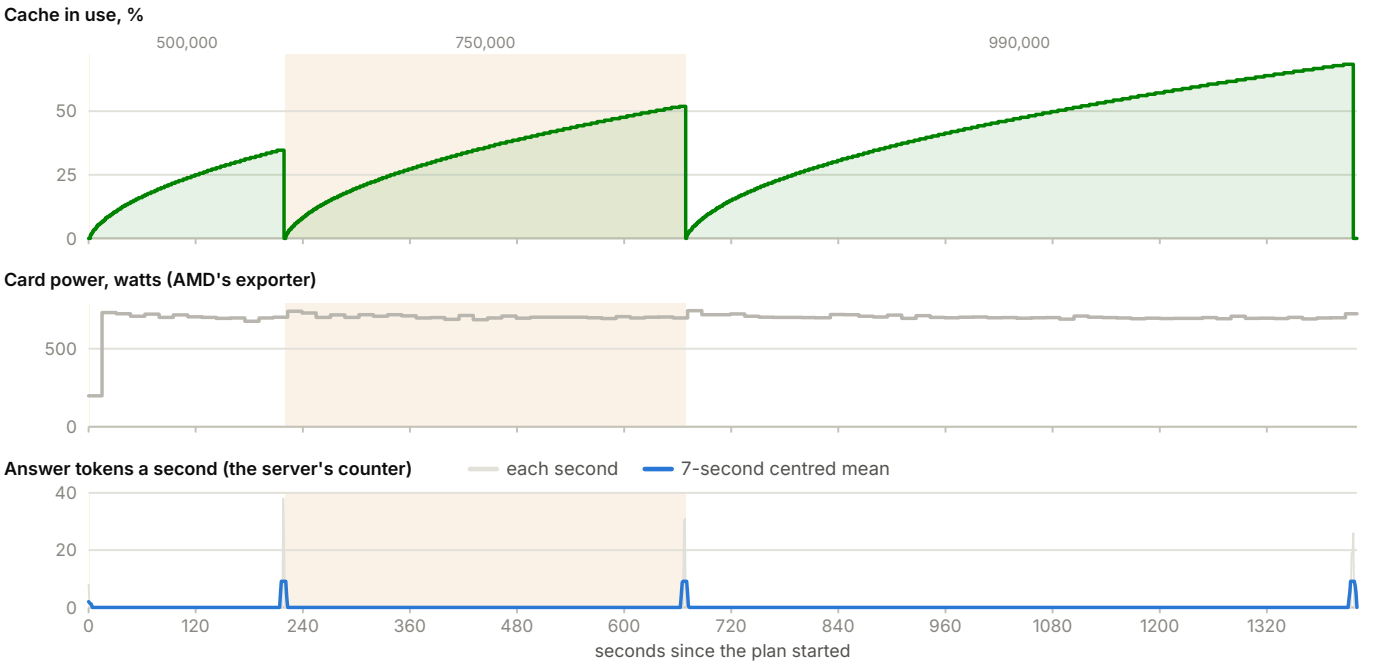

Figure 4 The whole plan. Top: the cache filling as each prompt goes in; it empties when the answer has been written. Middle: the card's power. Bottom: answer tokens, each second in grey and a 7-second centred mean in blue: 64 tokens at the end of each band, written in about two seconds. Shaded bands are the three prompts, labelled with their length.
Source: plan-context-ladder-1m/timeline.json (the server's gauges and counters every second; AMD's device exporter, which refreshes every 11 to 16 s)

The exact record

PROMPT TOKENS	FIRST TOKEN, S	THAT IS	READING, S (SERVER)	PROMPT TOKENS/S	WRITING, S (SERVER)	ANSWER TOKENS/S (SERVER)	BRACKET AT THE CLIENT	MS PER ANSWER TOKEN	CACHE IN USE, %
500,000	217.75	3 min 38 s	217.36	2,296	1.69	37.2	38.0 to 45.2	26.9	34.6
750,000	447.56	7 min 28 s	447.01	1,676	2.05	30.8	30.5 to 37.6	32.5	51.9
990,000	746.06	12 min 26 s	745.15	1,327	2.38	26.4	14.3 to 37.6	37.8	68.4
NOT A RUNG: THE WARM-UP REQUEST, 16 TOKENS ASKED BACK									
1,000	0.15	0.15 s	0.15	6,606	0.25	60.2	27.6 to 69.0	16.6	-

One row per request, from `summary.json` and `cross/ladder.json`. First token: at the client, from sending to the first text chunk. Reading and writing seconds: the server's own prefill and decode timers for that request. Prompt tokens a second: the prompt's tokens divided by the first-token time. Answer tokens a second: the answer's tokens less one, divided by the writing seconds. Cache: the most seen while the request played.

MEASURED

At 500,000 tokens the first token came after 3 min 38 s and the answer ran at 37.2 tokens a second. At 750,000: 7 min 28 s and 30.8. At 990,000: 12 min 26 s and 26.4.

KEEP IN MIND

Why the server's timer. The answers are random tokens, and a token that carries no visible text sends no chunk to the client. Counting chunks gives too low a speed; spreading all the tokens over the time from the first chunk to the last gives too high a one. At 990,000 tokens those two are 14.3 and 37.6: a bracket, not a measurement. The server's own decode timer says 26.4. At 500,000 it sits just below the bracket, because tokens without text at either end of an answer shorten the client's window.

Why twelve minutes: every bite costs more than the one before

Cache in use, %

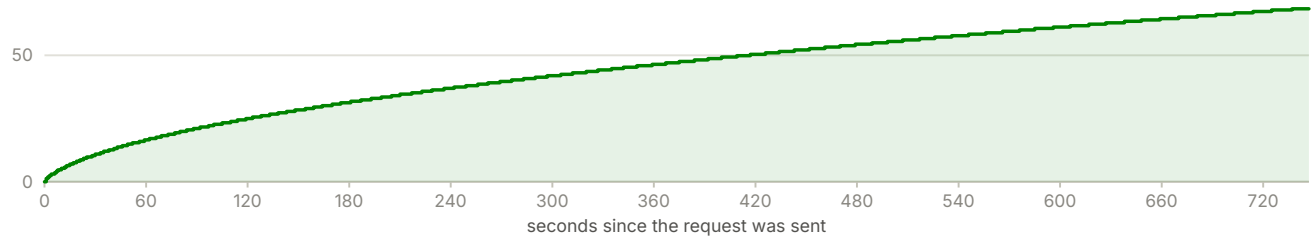

Figure 5 The 990,000-token request, second by second. The cache gauge climbs in steps as the bites of the prompt go in, and the steps get longer: the curve bends over all the way.
Source: plan-context-ladder-1m/timeline.json (vllm:kv_cache_usage_perc sampled every second), stage in-990000

Seconds for one bite

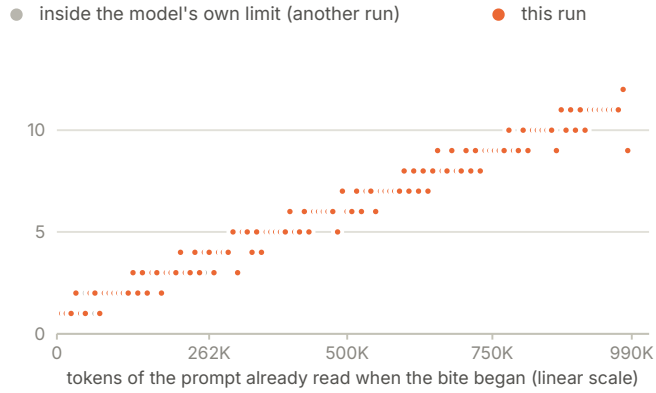

Figure 6 Reading. The seconds each bite of the 990,000-token prompt took, against what had been read before it. Whole seconds: the record has one sample a second. Grey: the 262,000-token prompt inside the model's own limit.
Source: timeline.json of both runs: the seconds between two moves of the server's work counter (est_tflops), which moves once per bite

Milliseconds for one answer token

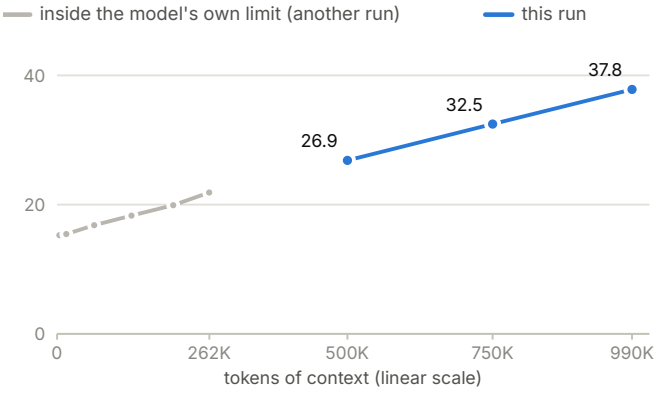

Figure 7 Writing. The time for one answer token against the context, on plain scales. The stretched prompts carry on the line of the ones inside the limit.
Source: computed from single_requests[].server_decode_s, this run and 20261125T215049Z-series-fa (cross/ladder.json)

MEASURED

The 990,000-token prompt went in as 121 bites of at most 8,192 tokens. The first three took 1.0 s each on average, the last three full ones 11.3 s. Its first token came 298.5 s later than that of the 750,000-token prompt: the last 240,000 tokens went in at 804 tokens a second.

COMPUTED

One answer token took 26.9 ms at 500,000 tokens of context and 37.8 ms at 990,000: 0.022 ms more for every 1,000 tokens. Inside its own limit the same kernel showed 0.026 ms (from 4,000 to 262,000 tokens, another run). Past the model's own limit a token of context costs no more than inside it.

Reading speed, by the part of the prompt

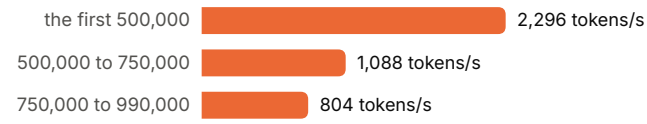

Figure 8 Reading once more, from the first-token times of the three requests: how fast the tokens that each longer prompt adds went in.
Source: computed from single_requests[].first_token_s of neighbouring rungs: tokens added divided by seconds added

READING, NOT MEASUREMENT

The same thing slows reading and writing: each new token looks back over all the tokens before it, and that costs time in proportion to how many there are. With 990,000 tokens the sum of those looks is what takes twelve minutes.

READING, NOT MEASUREMENT

vLLM's estimate counts too much attention for this model. For every bite already read it adds 105.6 million million operations to a bite, which is what attention over the context costs in 64 layers of this size (24 heads of 256 numbers). The model's configuration has full attention in 16 of its 64 layers.

Over the longest request the estimate is 1,097 TFLOPS, 84.4% of AMD's stated peak, and 94% of that work is attention over the context. Counted for 16 layers, the same arithmetic gives 324 TFLOPS, 24.9%. That last figure is my reading, not vLLM's: the card is busy, but not near its stated peak.

For scale: the same card at the model's own limit

Seconds before the first token

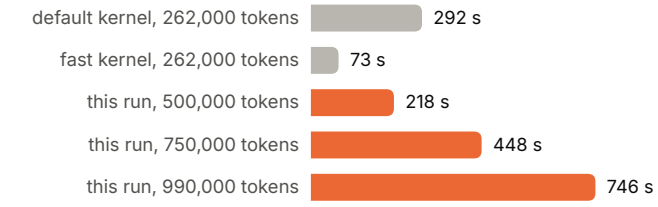

Figure 9 The wait for the first token: this run's three prompts, and in grey the full-context request of two other runs.
Source: summary.json single_requests[].first_token_s of this run, 20261125T215049Z-series-fa and 20261125T201346Z-series

Answer tokens written a second

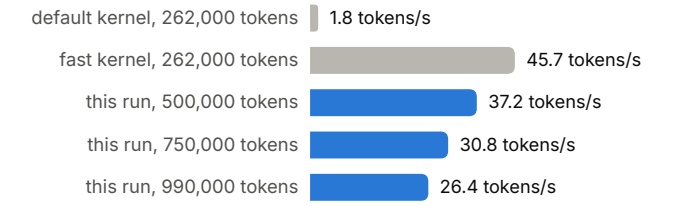

Figure 10 The same five requests: how fast the answer was written, by each server's own timer.
Source: summary.json single_requests[].decode_tok_s_server of the same three runs

COMPUTED

Against the same kernel at the model's full context: the longest prompt here is 3.7 times as long, its first token took 10.2 times as long, and writing kept 57.8% of its speed.

READING, NOT MEASUREMENT

On the default attention kernel this would not have been practical: it already needed 292 s and gave 1.8 tokens a second at 262,000 tokens. Here 500,000 tokens were read in 218 s.

The grey rows are other runs, there for scale and not like for like. The fast kernel's row differs in the context setting, the plan's seed and the size of the cache; the default kernel's row differs in the kernel as well.

What fits

Cache slots held at most, of the cache this server started with

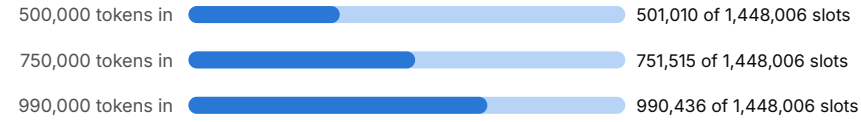

Figure 11 Each prompt against the cache this server started with. A slot holds one token, so a prompt takes about as many slots as it has tokens.
Source: summary.json plans.plan-context-ladder-1m.stages[].cache_in_use_max times server.kv_cache_tokens

MEASURED

The longest request held 68.4% of the 1,448,006 slots. One such request fits and two do not. Nor would the 500,000-token prompt fit beside it: together they need 103.0%.

Underneath

PROMPT TOKENS	CARD POWER, W (MEAN)	TFLOPS, VLLM'S ESTIMATE	% OF AMD'S STATED PEAK	CPU BUSY, %	ENGINE CORES	LONGEST GAP IN THE ANSWER, MS
500,000	673	1,021	78.5	10.8	2.05	29
750,000	704	1,074	82.5	10.9	2.06	65
990,000	702	1,100	84.5	10.8	2.05	79

Power is the mean of the exporter's readings over each request; new readings every 11 to 16 s. TFLOPS is vLLM's estimate from the model's shape, not a hardware counter, and the peak it is divided by is AMD's stated figure; the page before says why I read it as too high. CPU and engine cores are measured on the host. The longest gap is the longest silence between two chunks of the answer.

What I take from it

● With the model card's stretch setting and the fast kernel, one card reads a prompt of 990,000 tokens in 12 min 26 s and then writes at 26.4 tokens a second.

● The wait grows faster than the prompt: 3 min 38 s at 500,000 tokens, 7 min 28 s at 750,000.

○ Past the model's own limit a token of context costs no more than inside it: reading and writing go on slowing along the same lines.

● The longest request used 68.4% of this server's cache. One fits; two do not.

◑ vLLM's estimate for it is 1,097 TFLOPS, 84.4% of AMD's stated peak. That is too high for this model: counted for the 16 layers that have full attention it comes to 324 TFLOPS, 24.9% (computed).

◌ On the default kernel this would not have been practical.

Not results, and limits

NOT A RESULT

The 1,000-token request at the start is a warm-up, not a rung. It asked for 16 tokens back and got its first one after 0.15 s.

- **Speed and behaviour only.** Random-token prompts and answers of 64 forced tokens say nothing about the quality of an answer at this length.
- The position setting is the model card's own. I did not test what it does to answers inside the model's own limit either.
- One request at a time on an otherwise idle server, each prompt once. Writing speed is taken over 64 tokens, about two seconds.

● measured: read from a run file · ○ computed: arithmetic on measured values · ◑ estimate: vLLM's, from the model's shape · ◌ my reading: interpretation, not measurement

Words used

Context
all tokens a request holds: prompt plus answer so far. This model's native limit is 262,144.

YaRN
a way of stretching the model's sense of position so it can read past the context length it was trained for.

Prefill
the server reading the prompt. Every prompt token passes through the model once before the first answer token.

Decode
the server writing the answer, one token per stream per engine step.

First token
(time to first token) from sending the request to receiving the first answer token. It contains waiting in line and prefill.

Chunked prefill
a long prompt is read in bites across several engine steps, at most `max_num_batched_tokens` (8,192 here by default) per step for all requests together.

Cache
(KV cache) the GPU memory where the server keeps what the model computed for every token of every open stream, so it need not be recomputed.

Attention kernel
(`--attention-backend`) the GPU code that lets each token look back over the context. Four exist here: `ROCM_ATTN` (the default pick), `ROCM_AITER_FA` (AITER flash attention), `ROCM_AITER_UNIFIED_ATTN`, `TRITON_ATTN`.

AITER
AMD's library of GPU kernels for inference. Off by default in this install (`VLLM_ROCM_USE_AITER`).

TFLOPS
trillions of arithmetic operations per second. Here always vLLM's estimate from the model's shape (`--enable-mfu-metrics`), not a hardware counter. It counts attention in all 64 layers where this model has it in 16: 1.7% high for short prompts, over half too high from 128,000 tokens of context.

Every number in this document was read from these files at build time (sha256, first 12 characters):

`FACTS.json` 160f5ca0d495 · `GLOSSARY.md` 98323be796af · `cross/compute-estimate.json` b36121710947 · `cross/ladder.json` 3511faec1269 · `facts/box/20261125T192138Z/model-config.json` 191e0af23210 · `plans/context-ladder-1m.json` c0deac13164b · `runs/20261125T201346Z-series/summary.json` aab3940eb12e · `runs/20261125T215049Z-series-fa/plan-context-ladder/timeline.json` 6d9043e58e9a · `runs/20261125T215049Z-series-fa/summary.json` ec0c01913c37 · `runs/20261125T220624Z-ctx1m/plan-context-ladder-1m/timeline.json` f24b8af25366 · `runs/20261125T220624Z-ctx1m/summary.json` dc630cf24d26

65 values bound by path, 11 figures drawn from data. built 2026-11-26 02:38Z, explainer-pdf-system v2.



CREATED AND DIRECTED BY

Peter Styk

HPC & AI Platform Engineer

GPU inference, data and performance

linkedin.com/in/peterstyk · github.com/styk-tv

Questions about a run? Connect on LinkedIn.

LinkedIn