On the default kernel, speed collapses as the context grows

One request at a time, with a prompt that grows from 4,000 to 262,000 tokens. With the model's context full, the first token takes 292 seconds and the answer is then written at 1.8 tokens a second. A request that does not fit is refused at once.

Qwen3.8-27B · one AMD Instinct MI300X · vLLM 0.27.1 **Run** 20261125T201346Z-series
Changed nothing: baseline settings, the attention kernel vLLM picks by itself
Load 1,000 to 262,000 tokens in, 64 out, one request at a time, then two that do not fit

I sent one request at a time and made the prompt longer at every rung: 4,000 tokens, then 16,000, and so on up to 262,000, each time asking for 64 tokens back. On the attention kernel vLLM picks by itself, both speeds fell at every rung. With the model's context full, the first token came after 4 min 52 s and the answer was written at 1.8 tokens a second. Two requests that did not fit were refused at once.

1.8
answer tokens a second with 262,000 tokens of context; 43.6 with 4,000

4 min 52 s
before the first token at 262,000 tokens; 0.48 s at 4,000

898
prompt tokens read a second at 262,000, averaged over the prompt; 8,392.5 at 4,000

2 of 2
requests past the limit refused at once, with HTTP 400 and a plain message

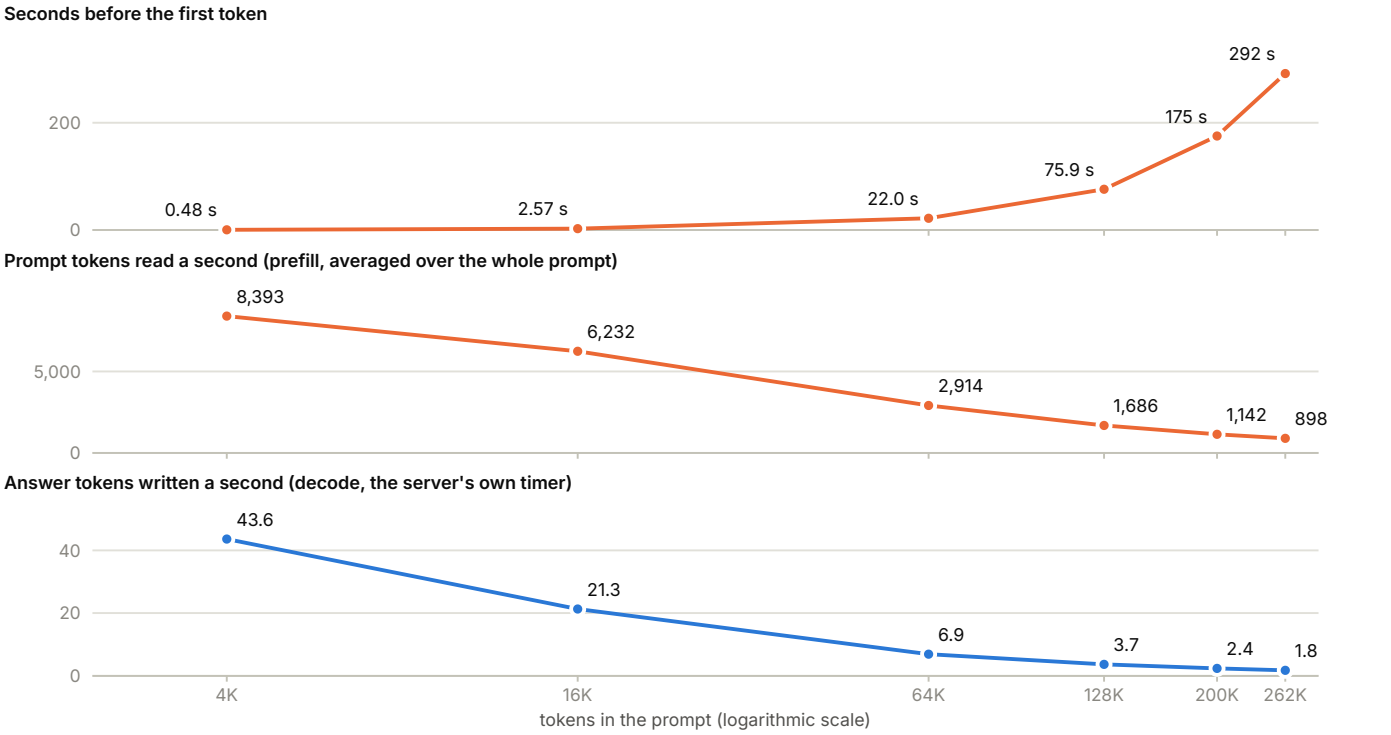


Figure 1 The six rungs. Top: the wait for the first token. Middle: how fast the prompt was read, averaged over the whole prompt. Bottom: how fast the answer was written. Reading is orange, writing blue. Every value is written at its point.
Source: summary.json plans.plan-context-ladder.single_requests[] (first_token_s, prefill_tok_s, decode_tok_s_server), read through cross/ladder.json

COMPUTED
From the first rung to the last the prompt grew 65.5 times, and the wait for the first token 611.3 times. Reading has no fixed price per token here: the more has been read, the more the next token costs.

MEASURED
Writing slowed as well: 43.6 answer tokens a second at 4,000 tokens, 21.3 at 16,000, 1.8 at 262,000. All 7 requests that fit were answered, and the server counted exactly the tokens I asked for.

The machine, and the three parts this test leans on

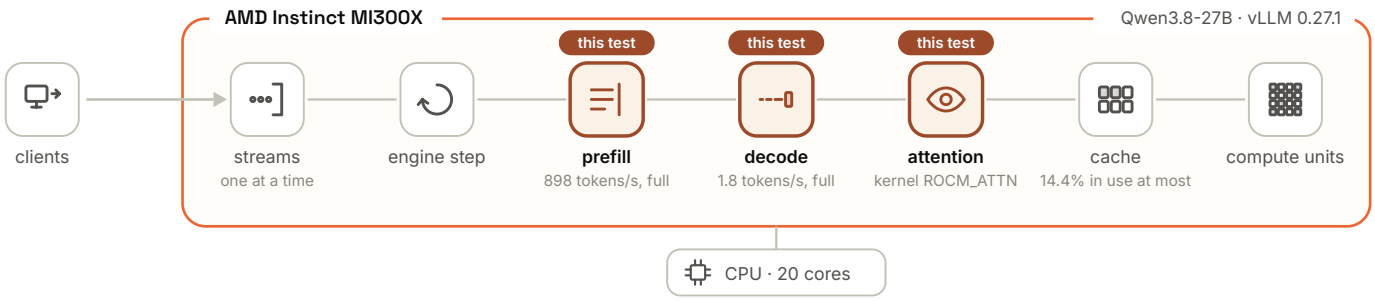


Figure 2 The card as every document in this series draws it. One request runs at a time, so nothing waits and the cache never fills: what is left to measure is how long prefill and decode take as attention gets more context to look back over. "Full" is the 262,000-token rung.
Source: FACTS.json; summary.json server.attention_kernel and plans.plan-context-ladder

One request, from prompt to answer



Figure 3 What one rung does. Each bite of the prompt, and later each answer token, looks back over everything already in the cache.
Source: loadgen.py; plans/context-ladder.json; FACTS.json server_defaults_on_this_card.batched_tokens_per_step

1. The plan sends 9 requests, one at a time: a prompt of 1,000 tokens, then 4,000, 16,000, 64,000, 128,000, 200,000 and 262,000, then two that cannot fit (262,100 and 263,000).
2. Every prompt is a list of random token ids, so its length is exact and nothing repeats. Every answer is forced to 64 tokens.
3. The next, longer prompt is sent only when the last token of the answer before has arrived. Nothing else runs on the server.
4. 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 limit of 262,144 tokens

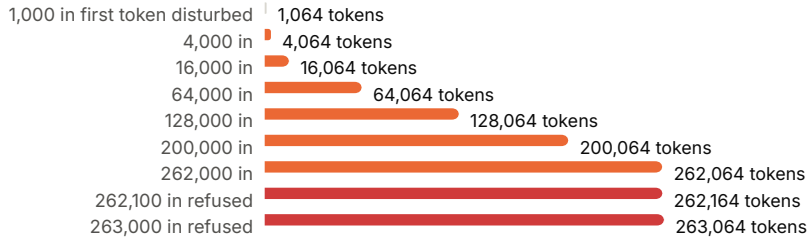


Figure 4 The plan. The prompt and the answer asked for count together, so 262,000 tokens in still fit and 262,100 do not. Grey: its first-token time is not a result. Red: refused.
Source: plans/context-ladder.json; summary.json server.max_model_len; cross/ladder.json (ended, is_result)

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	20261125T201346Z-series, started 2026-11-25T20:13:59Z, server ready after 94 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.

How each number is taken

- **First token**: at the client, from sending the request to its first text chunk. **Prompt tokens a second**: the prompt's tokens divided by that time, so an average over the whole prompt.
- **Reading and writing seconds**: the server's own prefill and decode timers for that request. **Answer tokens a second**: the answer's tokens less one, divided by the writing seconds.
- **The bracket at the client**: a token that carries no visible text sends no chunk, and these answers are random tokens. 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. I quote the server's timer and show those two only as a bracket. The server's figure can sit a tenth below it (16,000 and 64,000 here), because tokens without text at either end of an answer shorten the client's window.

Rung by rung: where the seconds go

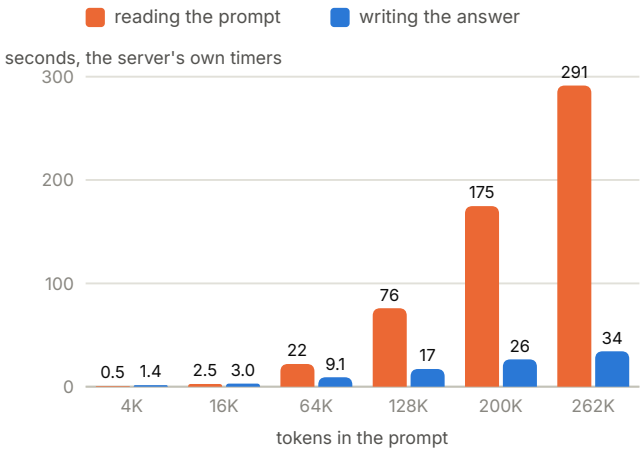


Figure 5 Seconds the server spent on each request: reading the prompt (orange) and writing 64 answer tokens (blue).
Source: summary.json single_requests[].server_prefill_s and .server_decode_s (the server's own timers per request)

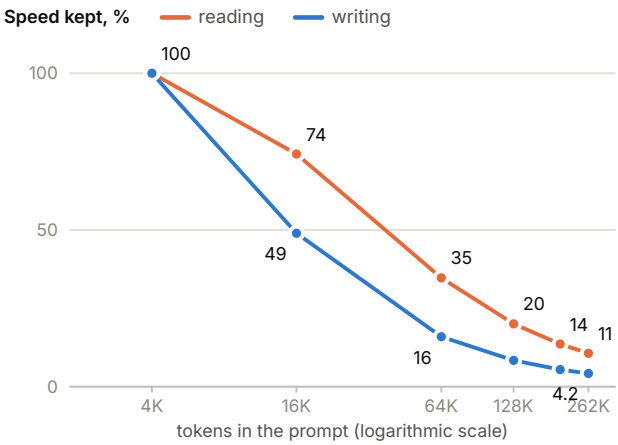


Figure 6 The same six requests as speeds, each as a share of its value at 4,000 tokens. Writing loses its speed sooner than reading.
Source: computed from single_requests[].prefill_tok_s and .server_decode_s

MEASURED

At 4,000 tokens the server spends more time writing 64 tokens (1.45 s) than reading the prompt (0.47 s). At 262,000 it reads for 291.4 s and then writes for 34.2 s.

COMPUTED

At 262,000 tokens reading keeps 10.7% of its speed at 4,000 and writing 4.2%. Writing is already down to 48.9% at 16,000.

The exact record

PROMPT TOKENS	FIRST TOKEN, S	READING, S (SERVER)	PROMPT TOKENS/S	WRITING, S (SERVER)	ANSWER TOKENS/S (SERVER)	BRACKET AT THE CLIENT	MS PER ANSWER TOKEN	CACHE IN USE, %
4,000	0.48	0.47	8,393	1.45	43.6	43.6 to 43.6	23	-
16,000	2.57	2.55	6,232	2.95	21.3	21.4 to 21.4	47	1.0
64,000	21.96	21.91	2,914	9.08	6.9	7.0 to 7.0	144	3.6
128,000	75.94	75.84	1,686	17.18	3.7	3.6 to 3.7	273	7.1
200,000	175.19	174.68	1,142	26.38	2.4	2.4 to 2.4	419	11.0
262,000	291.61	291.40	898	34.19	1.8	1.8 to 1.9	543	14.4
NOT A RESULT FOR THE FIRST TOKEN: SENT WHILE THE SERVER WAS STILL CLEARING THE PLAN BEFORE IT (DECODE IS NOT AFFECTED)								
1,000	7.99	0.13	125	1.04	60.4	26.3 to 61.4	17	0.2
PAST THE LIMIT: REFUSED AT ONCE (THE SERVER'S MESSAGE IS BELOW)								
262,100	HTTP 400 after 0.04 s	-	-	-	-	-	-	-
263,000	HTTP 400 after 0.04 s	-	-	-	-	-	-	-

One row per request, from `summary.json` and `cross/ladder.json`; how each column is taken is on the page before. The last column is the most cache seen while the rung played: one sample a second, so the shortest rung has none.

Past the limit

MEASURED

262,100 tokens in, 64 asked back: HTTP 400 after 0.04 s, before any of the prompt was read. The server said: "This model's maximum context length is 262144 tokens. However, you requested 64 output tokens and your prompt contains 262100 input tokens, for a total of 262164 tokens. Please reduce the length of the input prompt or the number of requested output tokens. (parameter=input_tokens, value=262100)"

263,000 tokens in: HTTP 400 after 0.04 s, the same message with its own figures.

Why: every token looks back over everything before it

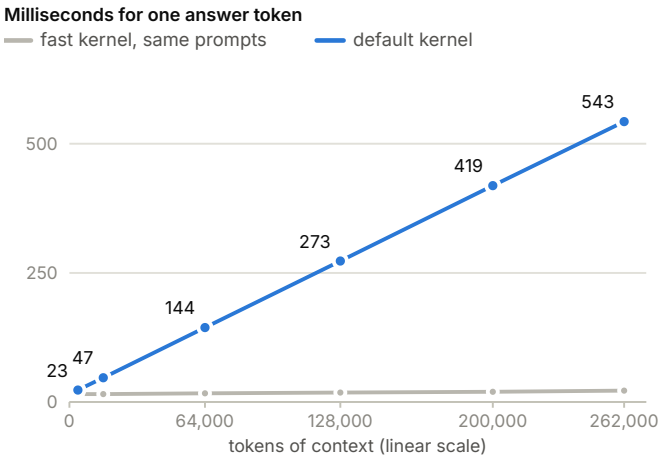


Figure 7 Writing. The time for one answer token against the context, on plain scales: a straight line. Grey: the same prompts on the fast kernel.
Source: computed from single_requests[].server_decode_s, this run and 20261125T215049Z-series-fa (cross/ladder.json)

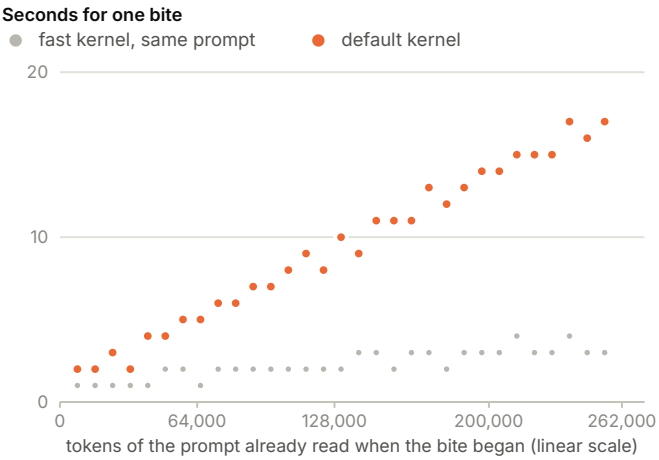


Figure 8 Reading. The seconds each bite of the 262,000-token prompt took, against what had been read before it. Whole seconds: the record has one sample a second. Grey: the same prompt on the fast kernel.
Source: plan-context-ladder/timeline.json of both runs: the seconds between two moves of the server's work counter (est_tflops), which moves once per bite

MEASURED

The 262,000-token prompt went in as 32 bites of at most 8,192 tokens. The first three took 2.3 s each on average, the last three 16.7 s. Its first token came 116.4 s later than that of the 200,000-token prompt: the last 62,000 tokens went in at 533 tokens a second.

COMPUTED

One answer token took 23 ms at 4,000 tokens of context and 543 ms at 262,000: 2.01 ms more for every 1,000 tokens of context. The four rungs between lie within 0.5% of the straight line through those two. On the fast kernel the same slope is 0.026 ms.

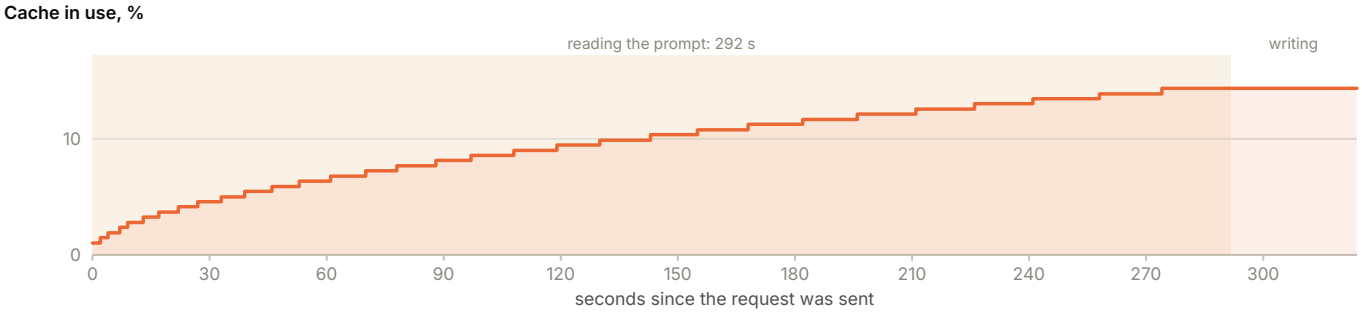


Figure 9 The 262,000-token request, second by second. The cache gauge climbs in steps as the bites go in, and the steps get longer: the curve bends over. The answer is written in the last band.
Source: plan-context-ladder/timeline.json (vllm:kv_cache_usage_perc sampled every second), stage in-262000

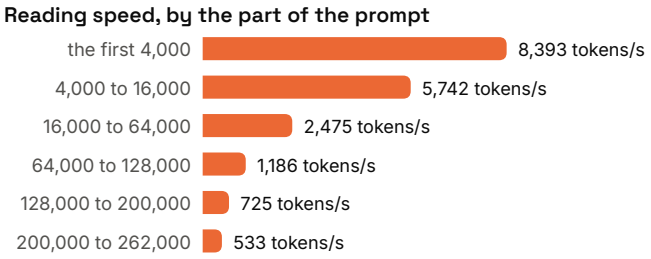


Figure 10 Reading once more, from the first-token times alone: how fast the tokens that each rung adds went in. The last 62,000 tokens of the longest prompt were read at about a sixteenth of the speed of the first 4,000.
Source: computed from single_requests[].first_token_s of neighbouring rungs: tokens added divided by seconds added

READING, NOT MEASUREMENT

Long context is slow because the 16 full-attention layers read the whole context at every step with the default kernel. The other 48 layers keep a state of fixed size and do not grow with the prompt. I base this on the straight lines above and on the fast kernel's flat ones: the same model, the same prompts, another kernel. I did not profile the default kernel at long context.

COMPUTED

A cost per token that rises in a straight line with the context makes a whole prompt cost its length squared. The 262,000-token prompt is 4.0 times as long as the 64,000-token one and took 13.3 times as long to read.

The same prompts on the other three kernels

Answer tokens a second, by kernel and prompt length

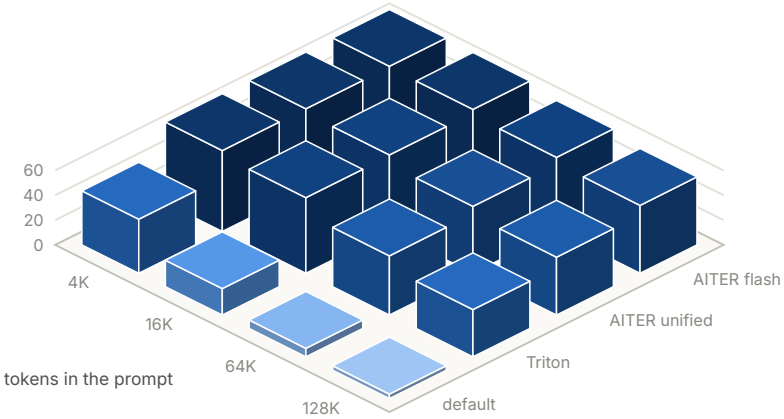


Figure 11 Writing speed on the four attention kernels this install offers, at the four rungs all of them played. A view only: the exact values are in the first matrix beside it. The default kernel is the front row.
Source: cross/ladder.json decode_tok_s_server (each run's summary.json)

	tokens in the prompt 4K	16K	64K	128K
AITER flash	65.8	64.8	59.8	54.8
AITER unified	65.2	61.7	53.9	46.7
Triton	65.3	61.0	47.8	38.1
default	43.6	21.3	6.9	3.7

darker = more: answer tokens a second

Figure 12 Writing: the same values as the view, exactly.
Source: cross/ladder.json decode_tok_s_server

	tokens in the prompt 4K	16K	64K	128K
AITER flash	8,979	7,194	6,604	5,234
AITER unified	8,329	7,200	4,297	2,799
Triton	8,239	6,312	3,257	1,973
default	8,393	6,232	2,914	1,686

darker = more: prompt tokens read a second

Figure 13 Reading, averaged over each prompt.
Source: cross/ladder.json prefill_tok_s

Each of the other three kernels ran on its own server: these settings plus `VLLM_ROCM_USE_AITER=1` and `--attention-backend`. They replayed the four lower rungs with the same seed. The default row is this run. In the matrices, AITER flash is `ROCM_AITER_FA`, AITER unified `ROCM_AITER_UNIFIED_ATTN`, Triton `TRITON_ATTN` and default `ROCM_ATTN`.

MEASURED

In writing the default kernel is the slowest of the four at every length. At 128,000 tokens it gives 3.7 answer tokens a second; Triton gives 38.1, AITER unified 46.7 and AITER flash 54.8.

MEASURED

In reading it is the slowest from 16,000 tokens up. At 4,000 the four are close and the default is second: 8,392.5 prompt tokens a second against 8,979.2 on AITER flash, 8,329.0 on AITER unified and 8,238.7 on Triton.

The whole ladder on the fast kernel

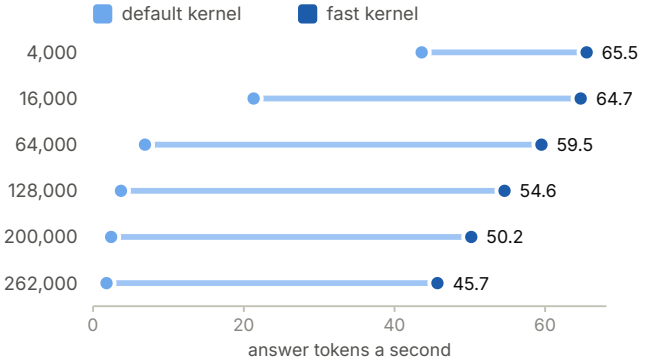


Figure 14 Answer tokens a second at each rung: this run, and the same plan on AITER flash attention (the fast kernel). The number is the fast kernel's.
Source: summary.json single_requests[].decode_tok_s_server of this run and of 20261125T215049Z-series-fa

PROMPT TOKENS	FIRST TOKEN, S: DEFAULT	FAST	SOONER BY	ANSWER TOKENS/S: DEFAULT	FAST	FASTER BY
4,000	0.48	0.45	1.0×	43.6	65.5	1.5×
16,000	2.57	2.21	1.1×	21.3	64.7	3.0×
64,000	21.96	9.74	2.2×	6.9	59.5	8.5×
128,000	75.94	25.00	3.0×	3.7	54.6	14.9×
200,000	175.19	48.38	3.6×	2.4	50.2	21.0×
262,000	291.61	72.85	4.0×	1.8	45.7	24.7×

"Faster by" divides the two servers' writing seconds for the same 64 tokens, which is exact where the rounded speeds are not. Multiples are cut to one decimal.

COMPUTED

With the context full, the fast kernel writes 24.7 times as fast (45.7 against 1.8 tokens a second) and reaches the first token 4.0 times as soon (1 min 13 s against 4 min 52 s).

KEEP IN MIND

Like for like: the same plan, the same card, one request at a time. What differs is the kernel, and AITER switched on for it. The fast kernel's server also starts with fewer cache slots (1,734,958 against 1,826,476), which does not matter here: one request uses 14.4% of the cache at most.

What the card was doing

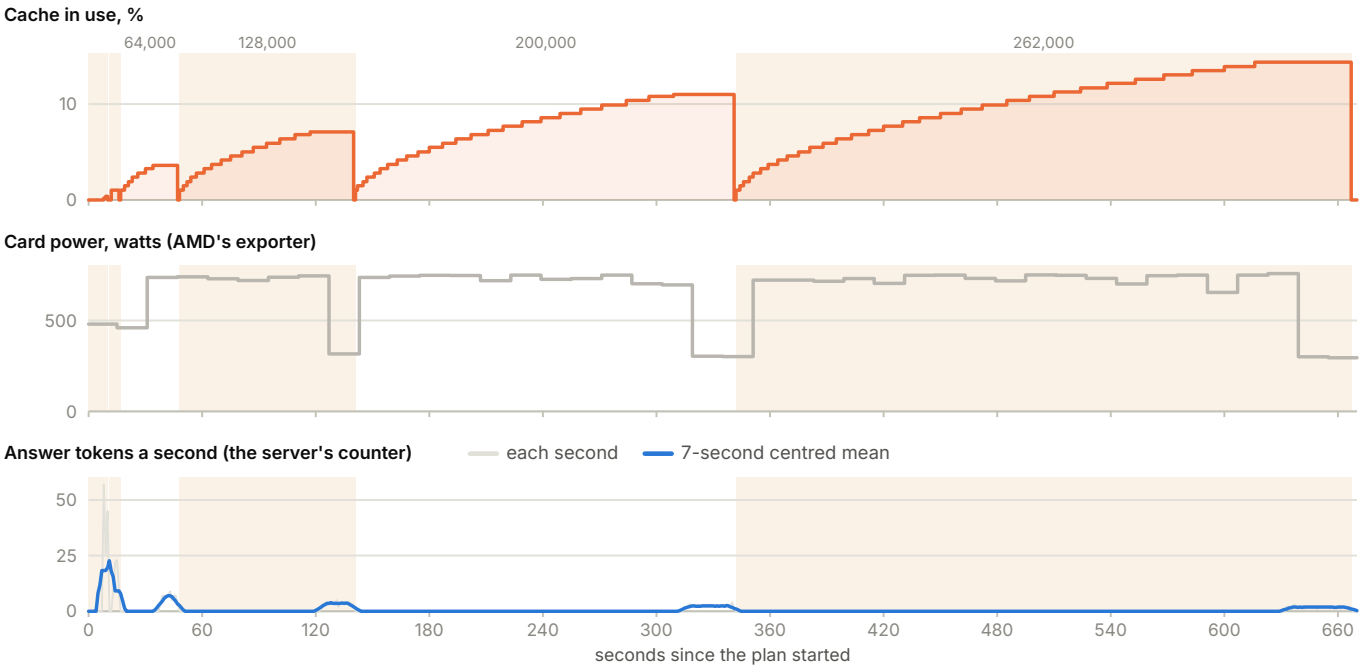


Figure 15 The whole plan, second by second. Top: the cache filling bite by bite at each rung. Middle: the card's power, which drops each time a long prompt has been read and the answer is being written. Bottom: answer tokens, each second in grey and a 7-second centred mean in blue. Shaded bands are the rungs, labelled with their prompt length.
Source: plan-context-ladder/timeline.json (the server's gauges and counters every second; AMD's device exporter, which refreshes every 11 to 16 s)

PROMPT TOKENS	WATTS WHILE READING	WATTS WHILE WRITING	TFLOPS, VLLM'S ESTIMATE	% OF AMD'S STATED PEAK	CPU BUSY, %	ENGINE CORES
64,000	706	-	206	15.8	10.6	2.00
128,000	733	316	207	15.9	10.6	2.00
200,000	732	302	206	15.8	10.6	2.01
262,000	729	297	206	15.8	10.6	2.00

The four rungs long enough for the exporter. Watts are means of its readings over each phase, leaving out the first 13 seconds of a phase because the exporter refreshes every 11 to 16 s. TFLOPS is vLLM's estimate from the model's shape over the whole request, reading and writing together; it is not a hardware counter, and the peak it is divided by is AMD's stated figure. CPU and engine cores are measured on the host.

MEASURED

At 262,000 tokens the card drew 729 W while it read the prompt and 297 W while it wrote the answer. With nothing running, the same card read 193 W just after this server started. Writing at long context adds about 104 W to an idle card; reading adds about 536 W.

ESTIMATE

◑ By vLLM's estimate the card did between 205.6 and 206.9 TFLOPS over each of the four long requests, about 15.8% of AMD's stated peak. On the fast kernel the same 262,000-token request reads 926 TFLOPS: the estimated work is the same, done in less time.

MEASURED

The CPU is not what waits: the machine was 10.6% busy and the engine process used 2.0 cores at every rung, as it does with any load.

READING, NOT MEASUREMENT

vLLM's estimate counts too much attention for this model. What it adds for every bite already read is attention over the context in 64 layers; the model's configuration has full attention in 16 of its 64. At 262,000 tokens 80% of the estimated work is that attention. Counted for 16 layers, the same arithmetic gives 82 TFLOPS, 6.2% of AMD's stated peak. My reading, not vLLM's figure.

READING, NOT MEASUREMENT

◌ While it writes at long context the card draws little more than when idle, although it reports itself busy. My reading is that the default kernel's decode step holds the card without using its width, so the wait is not arithmetic at full power. I did not profile it.

What I take from it

● On the kernel vLLM picks by itself, a request that fills the model's context waits 4 min 52 s for its first token and is then written at 1.8 tokens a second.

● Both speeds fall at every rung: reading from 8,392.5 to 898.4 prompt tokens a second, writing from 43.6 to 1.8.

○ One answer token costs 2.0 ms more for every 1,000 tokens of context, in a straight line.

● A request that does not fit is refused in a few hundredths of a second, with HTTP 400 and a message that names the limit of 262,144 tokens.

◌ The time goes to the full-attention layers on this kernel. Choosing the fast kernel removes most of it (page 5); it is not what vLLM picks when nothing is set.

Not results, and limits

NOT A RESULT

The first rung, 1,000 tokens, is not a result for the first token. Its 7.99 s were measured while the server was still clearing the 80 requests cut at the end of the plan before it. The server's own timers for that request read 0.13 s of prefill and 60.4 tokens a second of decode.

- Speed and behaviour only. Prompts are random tokens and answers are forced to 64 tokens, so nothing here says how well the model answers at this length.
- One request at a time on an otherwise idle server, each rung played once.
- With other streams running, a long prompt also holds up their answers. That is another test of this series (mixed lengths).

● 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

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.

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

Engine step
one pass of the model over a batch: one new token for every decoding stream, plus a bite of 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.

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`.

Linear-attention layers
48 of this model's 64 layers keep a fixed-size state per stream instead of looking back over every token; the other 16 are full attention.

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.

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.json` 2fe320742325 · `runs/20261125T201346Z-series/plan-context-ladder/requests.jsonl` 901cb4a70c6f · `runs/20261125T201346Z-series/plan-context-ladder/timeline.json` 7ee787305964 · `runs/20261125T201346Z-series/plan-smoke/timeline.json` 7230938b7ac0 · `runs/20261125T201346Z-series/summary.json` aab3940eb12e · `runs/20261125T211511Z-series-attn-rocm-aiter-unified-attn/summary.json` 12b96900e2e1 · `runs/20261125T212044Z-series-attn-triton-attn/summary.json` 6ecc28a27410 · `runs/20261125T212632Z-series-attn-rocm-aiter-fa/summary.json` 0272c72e9f27 · `runs/20261125T215049Z-series-fa/plan-context-ladder/timeline.json` 6d9043e58e9a · `runs/20261125T215049Z-series-fa/summary.json` ec0c01913c37

83 values bound by path, 15 figures drawn from data. built 2026-11-26 02:37Z, 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