rvLLM: GPU + TPU LLM Inference in Rust and JAX

Andy Norris (m0at)

https://github.com/m0at/rvllm

San Francisco, April 19, 2026

Abstract

rvLLM is an LLM inference engine spanning two hardware backends: a single-GPU Rust/CUDA path and a multi-chip TPU JAX/XLA path.

GPU (H100, Qwen2.5-7B, FP8). On a single NVIDIA H100 SXM 80 GB running Qwen2.5-7B-Instruct in FP8 E4M3 with greedy decoding, a single CUDA-graph replay per step, and real FA3 paged-prefill for prompt ingest, rvLLM v3 reaches **42,030 tokens/second at** $N = 512$ (decode + final RMSnorm + LM head + argmax) and a **63.8 ms time-to-first-token at** $N = 128$ for 16-token prompts. Comparing head-to-head at each batch size against vLLM 0.19 V1 on the same GPU with the same model and quant config: v3 measures **+13.8%** at $N = 128$ (22,069 vs vLLM's 19,399), **+22.8%** at $N = 256$ (34,364 vs 27,996), and **+16.4%** at $N = 512$ (42,030 vs 36,097) — consistently 14–23% faster at every batch size we tested. The earlier v2 stack peaked at 19,287 tok/s at $N = 128$ (SHA `eb9e247fd`) and is kept in-tree as an archive.

TPU (v6e-4, Gemma 4 31B, dual-path). A pure JAX + XLA implementation (∼500 lines of Python) runs Gemma 4 27B on a 4-chip TPU v6e pod slice with TP = 4. A *dual-path* architecture auto-switches based on `-max-ctx`: the *single-scan* path (≤32K context, bf16 KV cache, one `jax.lax.scan` over 60 layers with `cond` dispatch, 78.2 tok/s at B=1 512 ctx) and the *split-cache* path (>32K context, int8 KV cache, 10 groups of 6 layers, 50 sliding-window circular buffers + 10 global blockwise attention layers, 24.7 tok/s at 128K ctx, PPL 19.24). No compromise: 78.2 tok/s at short context, 24.7 tok/s at 128K. At $B = 768$ peak throughput is **13,943 tok/s** (2,681 tok/s/$). Measured head-to-head against vLLM on an H100 SXM 80 GB ($1.92/hr, FP8-Dynamic): rvLLM TPU is **17% faster at** B=1 (78.2 vs 66.9 tok/s) and **3.6× faster at peak** (13,943 vs 3,848 tok/s), with **34% better cost efficiency** (2,681 vs 2,004 tok/s/$). The split-cache path supports 128K context; the single-scan path provides the fastest short-context latency.

GPU path. v3 ships as a FlashAttention-3 SM90 paged-KV decode kernel (built from the Hopper source to `libfa3_kernels.so`, compiled with upstream's E4M3 paged instantiations), five FP8 linears driven by `cublasLtMatmul` with fused bias and residual epilogues, a set of custom fused PTX kernels (add+RMSNorm+quantize, RoPE + FP8 KV-cache write, SiLU·mul+quantize, final RMSNorm+quantize for LM head, argmax), and a graph-capture runtime that records all per-step kernel launches into a single `cuGraphLaunch`. Q, K, and V projections are fused into a single GEMM with bias epilogue (one call replaces three GEMMs and a bias kernel). The KV cache is FP8 E4M3 end-to-end: a custom `fused_rope_cache_fp8kv` kernel applies RoPE and writes FP8 pages; FA3's own E4M3 variants handle the dequantization inside the softmax. No megakernels: each launch is a recognizable composite.

The paper states the engine's correctness invariants and enumerates the structural bug classes that rvLLM is designed to eliminate: schedule-epilogue pairing mismatches in SM90 CUTLASS FP8 GEMMs (silent `CUDA_ERROR_ILLEGAL_ADDRESS` under graph replay only), captured metadata offsets clobbered by orthogonal upload paths, stale paged-KV block tables from CoW-only change detection, and silent correctness loss from omitting attention biases present in the model (Qwen2.5 `attention_bias=true`). Each is eliminated by a pinned variant, a frozen layout, a typed signal from the scheduler, or an explicit tensor load. Missing build artifacts refuse to start; no silent fallbacks exist in the FP8 dispatch. rvLLM is open-source under Apache-2.0.

1 Introduction

Inference serving for large language models is bandwidth-bound at decode batch sizes of practical interest. On an H100 running Qwen2.5-7B, a single decode step involves roughly 10–15 kernel launches per transformer layer, paged-KV attention over the sequence's current context, and a final projection to a 152,064-entry vocabulary. At a batch size of 128 and a sequence length of order 100 tokens, throughput is dominated by GEMM kernel efficiency, HBM traffic, and per-step launch overhead. On a TPU v6e pod slice the bottleneck shifts: XLA compiles the entire step into fused HLO; the performance lever is cache architecture and SPMD layout, not individual kernel selection.

vLLM [2] established the paged-KV cache design and continuous batching scheduler that define the modern serving baseline. CUTLASS 3.x [3] and FlashAttention-3 [5] supplied the kernel primitives that make Hopper-specific WGMMA and TMA accessible to non-Python callers. rvLLM composes these primitives behind a Rust runtime that owns CUDA graph capture, paged KV allocation, metadata packing, and sampling DtoH coordination in 31 crates (∼76K lines of Rust), with no Python interpreter in the serving path.

This paper documents rvLLM v3, which achieves **42,030 tok/s at** $N = 512$ and **63.8 ms TTFT at** $N = 128$ on a single H100 under a bench that re-uploads metadata each step and runs a real FA3 paged-prefill over the concatenated prompt (fair-comparison configuration, Section 7). vLLM 0.19 V1 runs on the same box at all three of our benched batch sizes; v3 measures 13.8% / 22.8% / 16.4% faster at $N = 128 / 256 / 512$ respectively. The previous v2 stack (archived at commit `eb9e247fd`, April 16, 2026) reached 19,287 tok/s at $N = 128$. v3's primary levers are: (i) fusing Q/K/V into a single GEMM where v2 runs three; (ii) routing all five FP8 linears (QKV, O, gate_up, down, lm_head) through `cublasLtMatmul`, whose per-shape heuristic explores many more algorithms than a 40-variant CUTLASS cache; (iii) fusing the QKV row-bias into the GEMM epilogue via `CUBLASLT_EPILOGUE_BIAS`, and the residual add into the O and down GEMMs via `beta`=1; (iv) f16 RoPE tables, eliminating f32 from the decode hot path; (v) an FP8 E4M3 KV cache built against FA3's upstream E4M3 paged instantiations, halving KV memory and enabling $N = 256/512$; (vi) a real multi-query causal FA3 paged-prefill entry point that replaces the 16-step eager faux-prefill, dropping TTFT 6–12× and lifting steady-state throughput once the prefill scratch sizing was corrected to `max_tokens`.

Sections 2–7 cover the GPU/Rust/CUDA path: Section 2 walks the v3 stack. Section 3 enumerates the kernel inventory. Section 4 gives an ordered trace of one decode step. Section 5 states the correctness invariants and ties each to a concrete failure mode observed during development. Section 6 preserves the v2 recovery history (9,531 → 19,287 tok/s in a one-day root-cause pass) and then describes the v3 session (551 → 42,030 tok/s across batch-size scaling). Section 7 reports the full GPU bench table. Section 8 presents the TPU path: Gemma 4 31B on a v6e-4 pod slice with the split-cache architecture, int8 KV quantization, and context scaling to 128K tokens.

2 The stack, every layer

2.1 HTTP, engine, scheduler

HTTP is an `axum` server exposing OpenAI-compatible `/v1/completions`, `/v1/chat/completions`, `/v1/embeddings`, and `/v1/responses`. Incoming requests are tokenized and placed on a lock-free MPSC queue consumed by a single worker thread; the worker is `!Send` by construction so a tokio runtime cannot accidentally schedule two concurrent GPU users.

The engine (`rvllm-v2::engine`) exposes two functions: `step_launch(diff)` enqueues all GPU work for one step (metadata HtoD, graph replay, argmax DtoH, event record) and returns

```
HTTP / OpenAI API          rvllm-api (axum)
Engine                     rvllm-v2::engine
                           step_pipelined() = launch + collect
                           double-buffered pinned argmax DtoH
Scheduler                  continuous batching
                           paged KV (block_size = 64)
                           page-boundary growth signal
Worker                     CUDA graph pool
                           35 pre-captured batch buckets
                           1 compute stream per worker
Runner                     1 HBM arena, pre-allocated slab
                           packed metadata (1 HtoD per step)
Layer                      11 kernel launches per layer
Kernels                    CUTLASS 3.x SM90 FP8 GEMM
                           FlashAttention-3 SM90 paged decode
                           custom fused PTX (norm / silu / rope)
                           cuBLAS HGEMM (LM head)
```

Figure 1: The rvLLM GPU stack (Rust/CUDA path). Each row runs in a single Rust crate; the DAG is enforced by a compile-time test that parses every `Cargo.toml` and rejects out-of-layer dependencies. The TPU path (Section 8) is a separate JAX codebase.

immediately; `step_collect()` waits on the previous step's DtoH event and returns the new tokens. There is exactly one codepath; the synchronous variant that tied `cuStreamSynchronize` to each step has been removed.

The scheduler (`rvllm-scheduler`) implements continuous batching and paged-KV allocation. A request moves through `Queued` → `Prefilling` → `Decoding` → `Finished`. The scheduler emits one `BatchPlan` per step, of exactly one variant (`Prefill`, `Decode`, or `Idle`); prefill and decode never coexist in the same step. A watermark-triggered preemption path re-queues the lowest-priority decoding requests when KV-cache block use exceeds 96%.

2.2 Worker, runner, layer

The worker (`rvllm-worker` / `rvllm-v2::worker`) owns one CUDA compute stream, one CUDA context, and a pool of pre-captured graph instances indexed by bucket size. Buckets are $\{1, 2, 4, 8, 16, 24, 32, \ldots, 256\}$. Graph capture happens during engine initialization, not during warmup.

The runner (`rvllm-v2::runner`) owns one HBM arena allocated via one `cuMemAlloc` at startup, sized for the worst-case workspace (sweep over all CUTLASS variants × all bucket sizes × all GEMM shapes), plus the KV cache, scratch tensors, and packed metadata buffer. A compile-time `GraphSafe` trait marks tensors whose lifetime outlives the capture region; `&mut HbmArena` is not in scope inside a `CaptureScope` closure, so runtime realloc is unrepresentable.

The layer (`rvllm-v2::layer`) is a pure function of (input, weights, KV cache slab, scratch, metadata handle). One decode layer at FP8 is eleven kernel launches:

1. `fused_add_rmsnorm_fp8_quant`
2. `CUTLASS FP8 GEMM (QKV)`
3. `fused_rope_kv_write`
4. `FA3 paged_decode`
5. `quantize_fp8_per_token (post-attention)`
6. `CUTLASS FP8 GEMM + residual (o_proj, v1)`
7. `fused_rmsnorm_fp8_quant`
8. `CUTLASS FP8 GEMM (gate_up)`
9. `fused_silu_mul_fp8_quant`

10. CUTLASS FP8 GEMM (down_proj)
11. residual_add_f16

Buffer aliasing is checked by the borrow checker: the layer takes `&mut Tensor<'a, f16>` for its output and `&mut LayerScratch` for scratch; two mutable borrows into one allocation fail to compile.

3 Kernel inventory

Every kernel has a pinned variant and a workspace contract. There are no dispatch fallback chains: each call site resolves to exactly one variant, determined at engine init from the autotune policy or pinned in source.

3.1 CUTLASS SM90 FP8 GEMM

CUTLASS 3.x templates drive the QKV, gate_up, down_proj, and fused o_proj+residual projections. Forty non-residual and ten residual-fused variants are instantiated, each parameterized by (TileShape, ClusterShape, KernelSchedule).

Variants cover tile shapes $\{64\times128, 128\times128, 128\times256\}$ in $M\times N$ with $K = 128$ or 256; cluster shapes $1\times1\times1$; and the `KernelTmaWarpSpecialized` (WS), `KernelTmaWarpSpecializedCooperative` (Coop), and their `FP8FastAccum` variants. Per-row activation scale and per-tensor weight scale are applied in the epilogue; the residual-fused kernel adds an `Sm90SrcFetch` fetch of the residual and a `TreeVisitor` computing $D = \alpha \cdot \mathrm{GEMM}(A, B) + C$.

Pairing constraint. The mainloop schedule must match the epilogue schedule. The residual-fused kernel uses `TmaWarpSpecializedCooperative` in its epilogue collective (`kernels/cutlass_fp8_gemm_res` line 100); any variant passing a non-cooperative mainloop with this epilogue causes `CUDA_ERROR_ILLEGAL_ADDRES` *only under graph replay* (the kernel runs cleanly outside a captured graph). The v2 runtime pins the o_proj residual path to variant `v1` ($128\times128\times128$ Coop/Coop matched); the v3 rewrite makes the mismatch a CUDA `static_assert`, rejecting the offending template pair at compile time.

Workspace contract. Each variant exposes `fp8_gemm_v<i>_workspace_size(M,N,K)`; the runtime's `max_workspace_size` query iterates all actually-callable variants across all bucket sizes for all GEMM shapes the model uses, and pre-allocates a single workspace slab accordingly. This eliminates the class of bug where the dispatch falls through to a variant whose workspace was never queried.

Autotune policy. A separate binary, `autotune-cutlass`, runs 67 HGEMM variants, 31 o-proj-residual variants, 32 gate-up-silu variants, 40 FP8 GEMM variants, and 10 FP8-GEMM-residual variants against every GEMM shape the model uses at every bucket size, reporting median of 10 timed runs after 3 warmups. The winning (shape $\rightarrow$ variant index) mapping is written to `policy.json`, which is a build artifact shipped alongside the binaries. The runtime does not consult `~/.cache` for policy; stale caches from prior deploys cannot influence dispatch.

3.2 FlashAttention-3 SM90 paged decode

`libfa3_kernels.so` is built at deploy time from the FlashAttention-3 Hopper source tree plus an `extern "C"` wrapper (`kernels/fa3_sm90_wrapper.cu`). Build time is roughly 10 minutes on an H100; the resulting shared object is 23 MB. The wrapper exposes `paged_decode` and `paged_prefill` entry points. Both expect KV laid out as [2, num_blocks, block_size, num_kv_heads, head_dim in one contiguous `Region`; the factor-of-two comes from interleaving K and V per block.

GQA is supported via num_heads/num_kv_heads; `head_dim = 128` is a hard gate (the kernel is specialized and refuses other head dimensions). `context_lens[i] == 0` is a valid padded-slot marker and yields zero output without touching `block_tables[i,*]`.

No PTX fallback. A prior PTX-only attention kernel existed in the tree. It has been removed from the dispatch; if `libfa3_kernels.so` is not present at engine init, the engine refuses to load. The build script ships with the repository.

3.3 Fused pre/post kernels

Eight custom CUDA kernels compile to PTX and are loaded by the runtime. Each fuses one recognizable composite. No megakernels exist in the dispatch path.

Kernel	Fuses	Launches
`embedding_gather`	token_ids → f16 hidden	$1 \to 1$
`fused_add_rmsnorm_fp8_quant`	residual + RMSNorm + quant	$3 \to 1$
`fused_rmsnorm_fp8_quant`	RMSNorm + quant	$2 \to 1$
`quantize_fp8_per_token`	post-attention act → FP8	$1 \to 1$
`fused_rope_kv_write`	RoPE + paged KV write	$3 \to 1$
`fused_silu_mul_fp8_quant`	SiLU + mul + quant	$3 \to 1$
`argmax`	f32 logits → i32 token	$1 \to 1$
`residual_add_f16`	$x + y$ in place	$1 \to 1$

Each fused kernel ships with a pure-Rust f32 reference implementation in tests; the PTX must match within cosine 0.999 for FP8 outputs and 10^{-5} absolute for f32 outputs, over a fuzz of shape and value distributions. The `residual_add_f16` kernel is deliberately *not* fused into the GEMM epilogue; prior attempts to fuse residual with the downstream projection hit either schedule-pairing crashes or a ∼24% regression from increased tile pressure.

Vectorization rule. Kernels that accept dimensions divisible by eight halves issue `uint4` 128-bit loads (eight halves per load) and `uint2` 64-bit FP8 stores (eight FP8 per store), with register caching between the absolute-max reduction pass and the quantize pass so each HBM element is read once. Alignment is checked by a `debug_assert` in the Rust binding; misalignment fails the launch loudly rather than producing `CUDA_ERROR_MISALIGNED_ADDRESS` under graph replay.

4 One decode step, in order

For a decode batch of N sequences at bucket size $B = \text{pad\_up}(N)$:

```
step_launch(diff):
  1. schedule(diff)                          CPU, O(N), zero allocs
  2. metadata pack + 1x HtoD                 packed i32 buffer:
                                             positions, context_lens,
                                             block_tables, slot_mapping
  3. graph_pool.replay(bucket = B)           one cuGraphLaunch:
       embedding_gather                       1 kernel
       for layer in 0..28:                    11 kernels x 28 = 308
         (as enumerated in Section 2.2)
       fused_residual_rmsnorm (final)         1 kernel
       cuBLAS HGEMM -> lm_head                1 kernel
       argmax                                 1 kernel
  4. cuMemcpyDtoHAsync argmax -> pinned[w]   4*N bytes (512 B at N=128)
     cuEventRecord event[w]
```

```
  5. return; w ^= 1                          double-buffer flip

step_collect():
  1. cuEventSynchronize event[r]             waits for step-1 DtoH
  2. read pinned[r][0..N]                    CPU reads new tokens
  3. scheduler.commit()                      mark tokens, free finished
```

Total kernel launches per decode step: $1 + 28 \times 11 + 1 + 1 + 1 = \mathbf{311}$, all captured into **one** `cuGraphLaunch`. Total DtoH per step: one $4 \cdot N$ byte copy of argmax token identifiers. Total HtoD per step: one packed metadata buffer, <100 KB at $N = 128$ with max_blocks$=129$.

The LM-head GEMM uses cuBLAS HGEMM (F16 in, F32 out) because CUTLASS FP8 autotune reported worse time on the $(N, 152064, 3584)$ shape at $M \geq 128$ than cuBLAS for the model tested. An FP8 LM-head path exists in `rvllm-v2` gated on the autotune cache's preferred variant; it is disabled by default for Qwen2.5-7B.

5 Correctness invariants

The engine enforces five invariants at build or startup time. Each is a response to a concrete failure mode observed during development.

5.1 No fallbacks

Rule. Missing autotune-policy entry for a shape panics with the shape. Missing `libfa3_kernels.so` refuses startup. Missing `libcutlass_kernels.so` refuses startup. The runtime does not read `~/.cache/rvllm/cutlass_autotune.json`; policy lives in the SHA-pinned `policy.json` shipped alongside the binary.

Failure mode this prevents. A prior deploy populated the on-box autotune cache with entries for an older CUTLASS catalog. A fresh build with the same HF-downloaded kernels inadvertently consulted the stale cache, resolving shape $(32, 152064, 3584)$ to a variant whose template no longer matched the `.so`'s symbol table, and ran the corresponding FP8FastAccum kernel under graph capture. Under graph replay, the kernel faulted with `CUDA_ERROR_ILLEGAL_ADDRESS`. SHA-pinning the policy artifact eliminates the failure class.

5.2 Single metadata-upload path

Rule. Metadata buffer layout is frozen per (bucket, max_blocks_per_seq). Captured graphs bind those exact device offsets. There is no non-padded upload variant. Prefill and decode use separate entry points that do not share offsets.

Failure mode. During development, two upload paths coexisted: `upload_metadata` (non-padded, sized to the actual request count) and `upload_metadata_padded` (sized to the captured bucket). On the first decode step after a prefill, the runtime's `last_meta_offsets` field carried the non-padded offsets from the prefill call, but the captured graph still read from the padded offsets. The diff-upload path wrote new token IDs, positions, and block-table entries at the non-padded indices; the captured kernel interpreted the bytes at the padded indices as if they were current data, producing a `context_lens` value of order 10^5, a `total_tiles` of 780, and a `block_tables` read 391 entries past the slice end. The read resolved to an uninitialized region of the meta buffer, which held bytes of a small integer interpreted as `phys_block`. Multiplied by the per-block stride 16384, this yielded a device address 10.2 MB before the nearest valid allocation—verbatim the `compute-sanitizer` report of the original crash.

The v3 rewrite removes non-padded upload entirely. The v2 engine tracks `last_padded_batch`:

`Option<usize>` and only takes the patch fast-path when it matches the requested bucket; otherwise a full padded upload happens.

5.3 Real block-change detection

Rule. The scheduler emits `ContinuedRequest::block_table_update: Option<Vec<BlockId>>` whenever a sequence's physical block list has grown since the last `mark_table_sent()`. The worker unions this with `diff.block_ops.copies` (copy-on-write events) to compute `has_block_changes` before each patch-decode upload.

Failure mode. An earlier version of the worker checked only `!diff.block_ops.copies.is_empty()`, which captures CoW events but *not* ordinary page-boundary growth. A sequence reaching the end of its current block and being allocated a new physical block generates no CoW event; the block-table entry at index $\lceil \text{context\_len}/\text{block\_size} \rceil - 1$ changes silently. The patch-decode path skipped the `block_tables` copy; the captured graph continued to read the stale block identifier, producing wrong KV-cache reads, garbage attention outputs, and garbage tokens. No crash. `ignore_eos=true` bench runs at output_len 512 still reported plausible throughput; correctness was degraded silently. The scheduler already emitted the needed signal; the worker simply did not consult it.

5.4 CUTLASS schedule/epilogue pairing

Rule. Mainloop schedule and epilogue schedule must match. A residual-fused kernel with `TmaWarpSpecializedCooperative` epilogue requires a cooperative mainloop schedule.

Enforcement. The v3 catalog uses a CUDA `static_assert` on the paired schedule types at template instantiation; the variant table generator writes Rust and CUDA sides together, so adding a mismatched variant fails compilation. In v2, the o_proj residual dispatch pins to the hand-verified variant `v1` ($128 \times 128 \times 128$ Coop/Coop), and the catalog has been audited for pairings—the offending variants are cutlass_fp8_gemm_v$\{0, 2, 3, 4, 6, 8, 11, 12, 13, 14\}$ and cutlass_fp8_gemm_residual_v$\{0, 2, 4, 5, 6\}$.

5.5 No `unwrap()` or `String` errors across crates

Rule. All errors crossing crate boundaries are `Result<T, RvllmError>`. The error variants carry structured context—stream identifier, kernel name, launch configuration, GEMM shape, workspace size needed vs given—not a stringified `DriverError`.

Failure mode. During the April 15 debugging pass, several downstream error paths presented the root-cause `cuGraphLaunch failed` as the opaque string `"dtoh event sync: DriverError(CUDA_ERROR_I ...)"`. The failing kernel name, stream identifier, bucket size, and kernel-launch arguments were discarded at the first `map_err`. Recovery required `compute-sanitizer` to re-derive what the error path should have carried.

6 Measured recovery, April 16, 2026

Three root-cause fixes, in three commits, took the engine from 9,531 tok/s to 19,287 tok/s at $N = 128$ in one working day. None were stopgaps; each eliminated a structural failure mode documented in Section 5.

Commit	Fix	**Δ**(N=128)
`6dabb76a5`	Track `last_padded_batch`; only patch metadata when bucket matches	crash → runs
`31716269d`	Re-enable fused FP8 o_proj + residual on variant `v1` (Coop/Coop)	+83%
`eb9e247fd`	Real block-change detection via `block_table_update` (not CoW-only)	+10%

In addition, `libfa3_kernels.so` was built on the deploy host (23 MB, once), eliminating a silent `.ptx` attention fallback path. All four changes taken together restore the engine to a configuration where every decode step uses the intended kernels, with no silent degradation in the dispatch.

7 Results

7.1 Experimental setup

All measurements in this section are on a single NVIDIA H100 SXM 80 GB (driver compatible with CUDA 12.4), running Qwen2.5-7B-Instruct from the public HuggingFace checkpoint, quantized to FP8 E4M3 at engine startup with per-tensor scale (clamp-ppm gate enforced). The v3 benchmark harness is `v3/crates/rvllm-bench`, driven by `RVLLM_*` environment variables; it runs 30 iterations per batch size after 5 warmup iterations and reports tokens per second as (iters × batch)/elapsed. Greedy decoding, CUDA graph captured once per bucket and replayed per iteration, all FP8 linears dispatched via `cublasLtMatmul`, `libfa3_kernels.so` built from the Hopper FlashAttention-3 source.

7.2 Head-to-head vs vLLM 0.19 V1 on the same H100

All figures below are each engine's own steady-state decode throughput metric. For vLLM it is the `Avg generation throughput` engine log line (prefill excluded). Same model, same FP8 E4M3 quantization, same CUDA-graph configuration, same 80 GB card.

N	vLLM 0.19	rvllm-v3	v3 Δ
128	19,399	20,841	+7.4%
256	27,996	31,178	+11.4%
512	36,097	**40,331**	**+11.7%**

Both engines scale through $N = 512$. A prior draft of this paper claimed vLLM could not fit $N = 256$/$N = 512$ on 80 GB. That was wrong — vLLM handles those batch sizes fine on the same box. v3's edge is a consistent 7–12% across the batch ladder, not a capacity-unlock step function.

What v3 adds architecturally. v3 ships FP8 E4M3 KV (1 byte/element, vs the FA3 default f16), which halves our per-layer KV allocation, and routes all five FP8 linears (QKV, O, gate_up, down, lm_head) through cuBLASLt with fused bias and residual epilogues. The per-token measurement above reflects both together; individual micro-benches at matched N show the cuBLASLt autotune was the dominant per-token win, while FP8 KV was essentially flat on per-token but halves the KV-cache memory footprint.

The vLLM measurement is taken on the same H100 SXM 80 GB box that runs v3, using the same `Qwen2.5-7B-Instruct` checkpoint and the same FP8 quantization setting, via `vllm bench latency -model Qwen2.5-7B -quantization fp8 -batch-size 128 -input-len 16 -output-len 512 -num-iters 3 -num-iters-warmup 1 -dtype float16`. The reported number is vLLM's engine log line `Avg generation throughput`, a steady-state decode metric that excludes prefill by construction. vLLM's total end-to-end latency for the 128-prompt, 512-output-token batch was 3.243 s.

What's 1:1 in this comparison. Same GPU, same model weights, same FP8 E4M3 quantization type, same batch size, CUDA graphs enabled in both. Both figures are decode tokens per second reported by each engine's own steady-state metric.

The three original caveats, and how v3's bench now addresses each. The initial v3 number at 22,559 tok/s raised three legitimate concerns about the fairness of a direct comparison against vLLM's serving measurement. We modified the v3 bench to address each one and re-ran:

1. **vLLM does real prefill** (16 input tokens × 128 prompts = 2,048 tokens of prompt processing before decode starts). vLLM's `Avg generation throughput` metric is decode-only, so this is mostly accounted for. *Fix (Phase F):* v3 now also runs a real multi-query causal FA3 paged-prefill under `RVLLM_REAL_PREFILL=1`: one attention call over the concatenated prompt (total_q = N × prompt_len) using varlen `cu_seqlens_q`. The pre-existing 16-step eager faux-prefill remains as a fallback.
2. **vLLM runs its scheduler every step.** At steady-state batch=128 it still checks request state, completions, and block-table management. The v3 bench previously replayed a frozen graph. *Fix:* the v3 bench now re-uploads `positions`, `slot_mapping`, and `context_lens` every timed iteration. v3 pays the same per-step HtoD cost that vLLM's scheduler pays for metadata management.
3. **vLLM's KV cache has real content from real prompts.** v3's had post-warmup garbage. *Fix:* the real FA3 paged-prefill populates the paged KV with legitimate rope+quant activations before the timed decode window, matching the state vLLM measures under. Metadata continues to advance through the decode window.

After all three fixes (and the FP8 KV migration and Phase F real-prefill path), v3 measures 22,069 tok/s at $N = 128$ (1.14× vLLM 0.19 V1), 34,364 tok/s at $N = 256$ (1.23×), and 42,030 tok/s at $N = 512$ (1.16×), on the same box. The 14–23% edge across the batch ladder confirms a genuine structural advantage (fused QKV, cuBLASLt autotune, fused epilogues, no Python dispatch, end-to-end FP8 KV), rather than a measurement artifact at a single point. TTFT is 63.8 ms at $N = 128$ with the real prefill path (§7.4).

Sources of v3's gain over v2. Four structural changes: (i) fused Q‖K‖V GEMM (one launch, $N = 4608$) versus v2's three separate Q/K/V GEMMs; (ii) `cublasLtMatmul` replacing hand-picked CUTLASS variants on all five linears (QKV, O, gate_up, down, lm_head); (iii) QKV bias fused into the GEMM epilogue via `CUBLASLT_EPILOGUE_BIAS`; (iv) O and down residual-add fused into the GEMM via `beta`=1 against the residual tensor. Each change removes a kernel launch without introducing a megakernel.

7.3 v3 session progression (at $N = 128$)

Step	tok/s	Δ
Eager (no graph)	551	—
+ CUDA graph capture	14,745	27×
+ f16 RoPE tables	15,537	+5%
+ q/k/v bias load + fused QKV	15,985	+3%
+ cuBLASLt + `EPILOGUE_BIAS` (QKV)	17,562	+10%
+ all 5 linears on cuBLASLt	22,496	+28%
+ FP8 E4M3 KV + final RMSnorm correctness	20,841	−7%
+ real FA3 paged-prefill (Phase F)	22,069	+6%

At $N = 128$ the FP8 KV migration costs a small amount of per-token throughput (FA3 dequant overhead at short contexts slightly outpaces the HBM bandwidth savings). The primary value of FP8 KV is the 2× memory halving, which at matched per-layer allocation leaves headroom for longer context lengths and — importantly for serving — fewer KV-allocator evictions under load. At $N = 256$ and $N = 512$ the per-token number scales up cleanly as GEMMs approach

HBM saturation (34,364 and 42,030 tok/s respectively with real prefill). Phase F's real prefill also lifted the $N = 128$ number back above the pre-FP8-KV high because the prefill scratch fixes (q_fp8 and residual sized to `max_tokens`) eliminated silent buffer truncation that had been hurting decode too.

7.4 Time-to-first-token (TTFT) with real prefill

Phase F replaces the 16-step eager faux-prefill with a single multi-query causal FP8 attention call via FA3's upstream hdim128 E4M3 paged instantiations (`fa3_sm90_paged_prefill_fp8`, exported from the same `libfa3_kernels.so` that serves decode). Q, K, and V are laid out as `[total_q, num_heads, head_dim]` indexed via a $[N{+}1]$ prefix-sum array `cu_seqlens_q`; a per-seq causal mask is applied so query t sees K positions $0 \ldots t$ within its own sequence. The fused rope+kv-write kernel that precedes attention writes FP8 K/V into the paged cache at the slots for *all* `total_q` tokens in one launch.

N	TTFT, real prefill	TTFT, faux-prefill	Speedup
128	63.8 ms	763 ms	12.0×
256	117.7 ms	833 ms	7.1×
512	131.7 ms	855 ms	6.5×

Measured with `RVLLM_PREFILL_LEN=16` (16 prompt tokens per sequence); t_0 is taken immediately before prompt ingest, t_1 at the first sampled token visible in pinned host memory (includes one decode step after prefill, the pinned DtoH, and a stream fence).

7.5 Where the cycles go (v2 baseline reference)

Per-kernel time measured on v2 at $N = 32$ (nsys profile); reproduced here for reference since v3's cuBLASLt path invokes different algorithms not directly characterized by v2's profiles:

- CUTLASS FP8 GEMM (fused residual, o_proj): 15.9–65.6 μs depending on bucket and variant.
- FA3 SM90 paged decode: 7.5 μs per layer at $N = 32$.
- fused RMSNorm + FP8 quant (per-token): 6.0–7.3 μs.
- fused SiLU*mul + FP8 quant: 14.3 μs.
- cuBLAS LM-head HGEMM: $\sim$483 μs on $(32, 152064, 3584)$.

In v3 the LM-head is FP8 GEMM via cuBLASLt with the activation pre-quantized, removing the HGEMM cost.

8 TPU: Gemma 4 31B on v6e-4

The TPU path is a pure JAX + XLA implementation: approximately 500 lines of Python, no custom kernels. XLA compiles the entire decode step to TPU machine code. The engine runs on a 4-chip TPU v6e pod slice (32 GB HBM per chip, 128 GB total) with tensor parallelism TP = 4 via `jax.sharding` SPMD.

8.1 Gemma 4 architecture

Gemma 4 27B (31B total parameters including embeddings) is a 60-layer transformer with dual attention: 50 layers use sliding-window attention (window size 1024 tokens) and 10 layers use global full-context attention. Architectural details: QK-norm on all layers; `k_eq_v` (shared K/V)

on global layers; GELU(`tanh`) activation; logit soft-capping $30 \cdot \tanh(x/30)$; tied input/output embeddings. Hidden dimension 5376; head dimensions 256 (sliding) and 512 (global).

The 60 layers are organized into 10 groups of 6: each group contains 5 sliding-window layers followed by 1 global layer. This regular structure is exploited by the split-cache design below.

8.2 Dual-path architecture

The key architectural innovation is a dual-path design that auto-switches based on `-max-ctx`. At ≤32K, the single-scan path uses bf16 KV cache with one `jax.lax.scan` over all 60 layers and `cond` dispatch for the fastest short-context latency (78.2 tok/s at 512 ctx). At >32K, the split-cache path uses int8 KV cache and treats sliding and global layers differently:

- **Sliding layers (50 of 60).** Each maintains a 1024-entry circular buffer. Regardless of total context length, each sliding layer's KV cache is $O(1024)$ — fixed memory, fixed compute. New tokens overwrite the oldest entry in the ring.
- **Global layers (10 of 60).** Each maintains a full-context KV cache. Attention is computed via blockwise attention with online softmax (BLOCK_K = 8192), enabling numerically stable processing of arbitrarily long contexts without materializing the full attention matrix.

The 10-groups-of-6 structure eliminates `jax.lax.cond` overhead: the loop body unconditionally runs 5 sliding layers then 1 global layer, with no dynamic branching. The KV cache is int8 with per-head scales (Section 8.8).

8.3 Results

Configuration	Batch	Context	ms/step	tok/s	KV type
Single-scan (cond dispatch)	1	512	12.79	78.2	bf16
Single-scan	1	2,048	∼14	∼70	bf16
Single-scan	1	32K	∼66	∼15	bf16
Split-cache (10 groups x 6)	1	64K	∼91	∼11	int8
Split-cache + blockwise	1	128K	40.56	24.7	int8
Single-scan, peak throughput	768	512	55.1	13,943	bf16

The dual-path architecture auto-switches at the 32K boundary: the single-scan path with bf16 KV cache is fastest for ≤32K context; the split-cache path with int8 KV cache enables 128K context.

Perplexity (WikiText-2, int8 KV split-cache): **19.24**. Cost: $5.20/hr for the v6e-4 pod slice, yielding **2,681 tok/s/$** at peak throughput.

8.4 Context scaling

Context	ms/step	tok/s	Architecture / KV
512	12.79	78.2	single-scan, bf16
2,048	∼14	∼70	single-scan, bf16
32K	∼66	∼15	single-scan, bf16
64K	∼91	∼11	split-cache, int8
128K	40.56	24.7	split-cache, int8

The dual-path architecture auto-switches based on `-max-ctx`. At ≤32K, the single-scan path with bf16 KV cache provides the fastest latency. At >32K, the split-cache path with int8 KV

cache enables 128K context. No compromise: 78.2 tok/s at short context, 24.7 tok/s at 128K.

8.5 Memory budget at 128K (per chip)

Component	Size
Weights (sharded, bf16)	7.75 GB
Sliding KV (50 layers × 1024 entries, int8)	50 MB
Global KV (10 layers × 128K entries, int8)	625 MB
Total	**~8.5 GB of 32 GB**

The split-cache leaves 23.5 GB per chip free at 128K context, enabling batch sizes up to 768 at shorter contexts.

8.6 Optimization progression

Stage	tok/s ($B=1$)	Δ
Nested `jax.lax.scan` (naïve)	25.6	—
Flat scan (groups of 6)	48.2	1.88×
+ int8 KV quantization	68.2	1.42×
+ fused `while_loop`	78.2	1.15×
+ $B=768$ batching	13,943	175×

The largest single gain is the nested-to-flat scan refactor, which eliminates per-group `jax.lax.cond` dispatch and lets XLA fuse the sliding-layer loop body into a single HLO block.

8.7 Comparison with vLLM on GPU (measured)

We measured vLLM on an H100 SXM 80 GB (vast.ai, $1.92/hr) serving the same Gemma 4 31B model (RedHatAI/gemma-4-31B-it-FP8-Dynamic, FP8) at max_ctx=2048. The full batch sweep:

Batch	vLLM GPU tok/s	vLLM GPU ms/step	rvLLM TPU tok/s
1	66.9	14.95	**78.2**
8	511.7	15.63	**584**
64	2,794	22.90	**4,220**
128	**3,848**	33.26	**6,831**
256	3,709	69.03	**10,536**
512	3,788	135.18	**12,932**
768	3,671	209.18	**13,943**

Key head-to-head results:

- **B=1:** rvLLM TPU 78.2 tok/s vs vLLM GPU 66.9 tok/s — TPU 17% faster at single-user latency (single-scan architecture, bf16 KV, 12.79 ms/step vs 14.95 ms/step).
- **Peak throughput:** rvLLM TPU 13,943 tok/s at B=768 vs vLLM GPU 3,848 tok/s at B=128 — TPU 3.6× faster. vLLM GPU saturates at B=128 and throughput declines beyond that; TPU scales near-linearly to B=768.
- **Cost efficiency:** TPU at $5.20/hr yields 2,681 tok/s/$; GPU at $1.92/hr yields 2,004 tok/s/$. TPU is 34% more cost-efficient at peak throughput.

- **128K context:** Only the TPU split-cache architecture supports 128K (24.7 tok/s). vLLM GPU was tested at max_ctx=2048.

TPU batch scaling numbers are from the single-scan bf16 KV architecture (512 ctx). The dual-path engine auto-switches to split-cache int8 KV for contexts >32K.

Prior vLLM-on-TPU reference. Google's published vLLM TPU benchmark reports Gemma 3 27B at 1,126 tok/s on the same v6e-4 hardware in bf16. rvLLM achieves 13,943 tok/s at $B = 768$, a 12.4× difference. The comparison is not apples-to-apples: the models differ (Gemma 4 vs Gemma 3), the quantization differs (int8 KV vs bf16), and the batch sizes likely differ.

8.8 Int8 KV cache

The KV cache uses int8 quantization with per-head scales: each attention head's K and V tensors are independently scaled to the int8 range. Gemma 4's QK-norm bounds the magnitude of query and key vectors before the dot product, making int8 quantization safe — the norm prevents outlier activations from dominating the quantization range.

Perplexity (WikiText-2) improved from 25.51 (bf16 KV, nested scan) to 19.24 (int8 KV, split-cache). The improvement is not from int8 itself but from the architectural changes that accompanied it: the split-cache correctly handles the sliding/global distinction, and the flat scan eliminates numerical artifacts from the nested `jax.lax.cond` path.

8.9 API server and chat client

The TPU engine is served via an OpenAI-compatible API endpoint (FastAPI), supporting `/v1/chat/completions` with streaming. A native Rust `egui` dual-pane chat client connects to the endpoint, providing a racing-style interface that displays token generation in real time. The client runs on the local machine; the TPU engine runs on the pod slice.

8.10 GPU: Gemma 4 31B on H100

The GPU engine runs the same 31B Gemma 4 model on a single H100 SXM 80 GB using Rust + CUDA. Weights are FP8 E4M3 with per-channel scales (from the RedHatAI FP8-Dynamic checkpoint); the KV cache and attention use F16 for precision. Three rounds of kernel fusion reduced CUDA graph nodes from 1776 to ∼1400 (21% reduction).

Perplexity. Three weight configurations were validated against the HuggingFace BF16 reference (PPL 19.62) and TPU int8 baseline (PPL 19.24):

Weight path	KV	PPL	tok/s
FP8-Dynamic + fused channelscale	F16	**13.53**	39.6
BF16 split QKV FP8	F16	17.96	37.9
F16 (no FP8)	F16	19.79	37.9
HF BF16 reference	–	19.62	–
TPU int8 reference	int8	19.24	–

Batch scaling. CUDA graph capture (∼1400 nodes) eliminates kernel launch overhead, yielding ∼4.7× speedup at B=1 (52 vs 11 tok/s eager). Throughput scales near-linearly through B=32, saturating as FP8 tensor cores become compute-bound:

Batch	tok/s	ms/step	Scaling	Eff.
1	52	19.2	1.0×	100%
4	229	17.5	4.4×	110%
8	452	17.7	8.7×	109%
16	900	17.8	17.3×	108%
32	1,723	18.6	33.1×	104%
64	3,097	20.7	59.6×	93%
128	5,114	25.0	98.3×	77%
256	6,897	37.1	132.6×	52%
512	7,943	64.5	152.8×	30%

Kernel fusion. Three fusion passes reduced graph nodes from 1776 to ∼1400: (1) `f32_to_bf16` + `rmsnorm_inplace` + `vector_add` fused into `fused_norm_add_residual` (−240 nodes); (2) `scale_cols_f32` fused into the norm+add kernel as `fused_norm_add_residual_f16` with channelscale in F32 (−120 nodes); (3) `residual_scale_f16` fused into the post-FF norm+add (−60 nodes).

Key engineering decisions. (1) Per-channel FP8 weight scales are fused into the post-GEMM norm+residual kernel, applying channelscale in F32 before RMSNorm. This avoids the separate `scale_cols_f32` pass and preserves precision (channelscale before BF16 truncation). (2) The KV cache stays in F16 because FP8 KV with a fixed scale destroyed Gemma 4 attention precision; F16 KV is critical for quality. (3) Per-layer KV cache sizing: sliding layers (50/60) use only 32 blocks (1024 tokens / 32 = 32), while global layers get the full block budget. This reduces KV memory ∼5× and enables 128K context on a single 80 GB GPU.

9 Related work

vLLM [2] introduced the paged-KV cache and continuous batching patterns that rvLLM's scheduler implements. The paged KV layout, block-manager structure, and preemption semantics follow vLLM's conventions closely; the departure is the Rust runtime, the graph-replay discipline, and the explicit rejection of the dispatch fallback chain.

FlashAttention-3 [5] supplies the SM90 paged-decode kernel; rvLLM links directly against a `.so` built from the Hopper source tree and does not re-implement the attention kernel. The `.so` build, variant selection, and workspace contract are rvLLM's responsibility.

CUTLASS [3] supplies the FP8 GEMM templates. rvLLM generates the variant catalog, enforces the schedule-pairing rule, and ships a per-shape autotune policy as a build artifact; the CUTLASS library itself does not enforce pairings or manage workspaces across a runtime.

TensorRT-LLM [4] and SGLang [6] occupy the same space but both assume a CUDA-Python host process. rvLLM's GPU path demonstrates that a GPU-serving engine can be written entirely in a systems language with no Python interpreter in the critical path, and that doing so exposes the kernel-level correctness invariants that a CUDA-Python host can paper over.

On TPU, vLLM's Pallas backend and JetStream [1] are the primary open-source baselines. rvLLM's TPU path takes a different approach: pure JAX with no custom Pallas kernels, relying entirely on XLA's compiler to generate efficient TPU code. The split-cache architecture (separate sliding-window and global-attention KV stores) is, to our knowledge, the first implementation that exploits Gemma 4's dual-attention structure for inference-time memory and compute savings.

10 Limitations and future work

The GPU engine at `eb9e247fd` is single-GPU only. Multi-GPU tensor parallelism exists in a sibling crate but is not wired into the v2 FP8 path. Pipeline parallelism is not implemented on the GPU side. The TPU path supports multi-chip tensor parallelism (TP = 4 on v6e-4) but has not been tested on larger pod slices. The GPU model catalog validated end-to-end on the FP8 path is Qwen2 family, Llama 3 family, Mistral 7B, and Gemma 2; a GPU FP8 path for Gemma 4 is in progress (scale calibration under development). The TPU path currently supports Gemma 4 only with a dual-path architecture that auto-switches based on `-max-ctx`: single-scan with bf16 KV (≤32K, fastest short-context latency) and split-cache with int8 KV (>32K, 128K context support). Peak throughput numbers reported in this paper use the single-scan architecture at 512 context. Other supported architectures compile but have not been validated against HuggingFace reference on these versions. Speculative decode, LoRA serving, and tool-calling guided decoding are present in the Python-free API layer but outside this paper.

The v3 rewrite consolidates the engine into 16 crates with a DAG-enforced dependency graph and per-crate LoC budgets, and makes each of the five invariants in Section 5 either a compile error, a type-level rejection, or a single-source-of-truth runtime check. The CUDA `static_assert` on schedule pairings, the removal of the second metadata upload path, the `GraphSafe` marker trait rejecting `&mut HbmArena` inside `CaptureScope` closures, and the SHA-pinned `policy.json` artifact each eliminate one of the three bug classes documented above. v3 is tracked under `v3/SPEC.md` and `v3/IMPL_PLAN.md` in the repository.

11 Conclusion

rvLLM demonstrates that high-performance LLM inference engines can be built outside the Python-hosted monoculture, across two distinct hardware backends.

GPU. The v3 Rust/CUDA engine achieves **42,030 tok/s** at $N = 512$ (1.16× vLLM 0.19 at the same batch; 1.14–1.23× consistently across $N = 128/256/512$) and **63.8 ms TTFT at** $N = 128$ (real FA3 paged-prefill; 12.0× faster than the 16-step eager faux-prefill stand-in it replaces) serving Qwen2.5-7B in FP8 on a single H100, with 339 kernel launches per decode step compressed into one `cuGraphLaunch` replay. The archived v2 stack (commit `eb9e247fd`) reached 19,287 tok/s at the same batch on the same GPU.

TPU. The JAX/XLA engine reaches **78.2 tok/s** at $B = 1$ (single-scan, bf16 KV, 512 context) and **13,943 tok/s** at $B = 768$ serving Gemma 4 31B on a v6e-4 pod slice. A dual-path architecture auto-switches based on context length: the single-scan path with bf16 KV cache (≤32K) provides the fastest short-context latency; the split-cache path with int8 KV cache (>32K) — 50 sliding-window layers in fixed-size circular buffers, 10 global layers with blockwise attention — scales to 128K context (24.7 tok/s). No compromise: 78.2 tok/s at short context, 24.7 tok/s at 128K. Measured against vLLM on H100 SXM 80 GB (FP8-Dynamic, $1.92/hr): TPU is 17% faster at B=1 (78.2 vs 66.9 tok/s), 3.6× faster at peak (13,943 vs 3,848 tok/s), and 34% more cost-efficient (2,681 vs 2,004 tok/s/$). At $5.20/hr the engine delivers 2,681 tok/s/$.

The path to the GPU v3 number was mechanical, not magical: capturing the layer loop into a CUDA graph alone moved the engine from 551 tok/s (eager) to 14,745 tok/s (replayed). Fusing Q/K/V into a single GEMM, loading attention biases that v2 had silently omitted, routing every FP8 linear through `cublasLtMatmul` with bias and residual epilogues, linking FA3's upstream hdim128 E4M3 paged instantiations for FP8 KV, and wiring a real multi-query causal paged-prefill entry point took the engine the rest of the way. The TPU path followed a similar trajectory: nested scan to flat scan to int8 KV to fused `while_loop`, each step guided by XLA's HLO profile rather than individual kernel timings.

The three SM90 / graph-capture failure modes documented in Section 5 — producing either silent correctness bugs or `CUDA_ERROR_ILLEGAL_ADDRESS` only under graph replay — are made unrepresentable in v3's type system or caught at compile time. A fourth class of silent bug surfaced during v3 development: omitting model components assumed by the architecture (here, Qwen2.5's attention biases). The invariant added for v3 is that every tensor listed in the model's safetensors index must be consumed by the loader or the engine refuses to start.

Addendum (June 2026): Gemma 4 31B single-user decode on H100

This addendum records a measured session (commits `f70b4bf`–`0d5f276`, 2026-06-09/10) on the Gemma 4 31B H100 path that postdates the body of this report, and corrects one analysis the April work relied on.

Results. Single-user (batch-1) full-generate decode moved from 44.9 to **63.0 tok/s** at short context and from 14.0 to **61.4 tok/s** (4.4×) at the production-shaped context (1,200–1,500 tokens, FP8 KV); n-gram speculative decoding with a graph-captured batched verify reached **83.9 tok/s** on real text (accept rate 0.42–0.56 per draft, K=4). The changes: (i) routing $M \leq 16$ FP8 GEMMs through `cublasLtMatmul` instead of the CUTLASS channelscale kernel, which node-level tracing showed running M=1 at only ∼51% of HBM (+27% end-to-end); (ii) a split-KV, GQA-grouped rewrite of the FP8 paged-decode attention kernel (551 μs → 15.7 μs per sliding-window call at the full 1,024-token window; 171× at 8K context; parity $\leq 4.9\times10^{-4}$); (iii) graph-capturing the speculative verify forward per chunk size. Deployed to production 2026-06-10 and measured through the live API: 3.38× short-prompt decode, 1.78× at 3K-token prompts.

Correction. The April batch-1 analysis attributed ∼12 ms/step to inter-node dispatch gaps inside the captured graph, because the per-kernel sum from a graph-level `nsys` trace (∼10.3 ms) was compared against the 22 ms step wall. Node-level tracing (`-cuda-graph-trace=node`) shows the graphed step spent ∼21.3 ms *in kernels* with only ∼1.1 ms of true gap; the difference was GEMM inefficiency and the serial-token FP8 attention kernel, both fixed above. The persistent-megakernel direction motivated by the dispatch-gap reading was independently built and refuted by measurement (26 vs. 45 tok/s). One operational finding worth a sentence: the production "FA3" shared object was discovered to be the Ada (sm_89) fallback kernel set under the FA3 filename — the loader's silent symbol-probe fallback has been replaced with a hard error on sm_90.

References

[1] Google. JetStream: Throughput and memory optimized engine for LLM inference on XLA devices. https://github.com/google/JetStream, 2024.

[2] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023. doi: 10.1145/3600006.3613165.

[3] NVIDIA Corporation. CUTLASS: CUDA templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass, 2024.

[4] NVIDIA Corporation. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM, 2024.

[5] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and exact attention with asynchrony and low-precision, 2024.

[6] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Siyuan Cheng, Jeff Huang, Simon Bi, Ying Sheng, Joseph E. Gonzalez, and Ion Stoica. SGLang: Efficient execution of structured language model programs, 2024.