Open-Jev-27B Inference Speedup Report

An inference backend for **Open-Jev** [1] (Open-Jev contributors). Model: Open-Jev-27B-v1.1 [2] on Qwen3.8-27B [3], bf16, not quantized. Hardware: one NVIDIA B300. Code, per-task results and logs: github.com/lyuyiqi/open-jev-fast (MIT). Author: Yiqi Lyu. Advisor: Zhaoran Wang. 2026-09-27.

Summary: for the example request (3 questions / 7 candidates / 539 tokens), one inference takes **20.1 ms**, vs **257.0 ms** on Open-Jev's default PyTorch path (**12.8×**) and **102.9 ms** with FLA kernels (**5.1×**). On the 231 JevBench tasks, mean latency went from **703 ms to 51 ms (13.7×)**, P50 171 → 24 ms, slowest task 16.0 s → 0.58 s. Accuracy 198/231 → 197/231; the one task that changed is a coin flip whose original probabilities are 0.504 vs 0.496 (numerical noise). It runs locally as a drop-in replacement for Open-Jev's own local server (same request format).

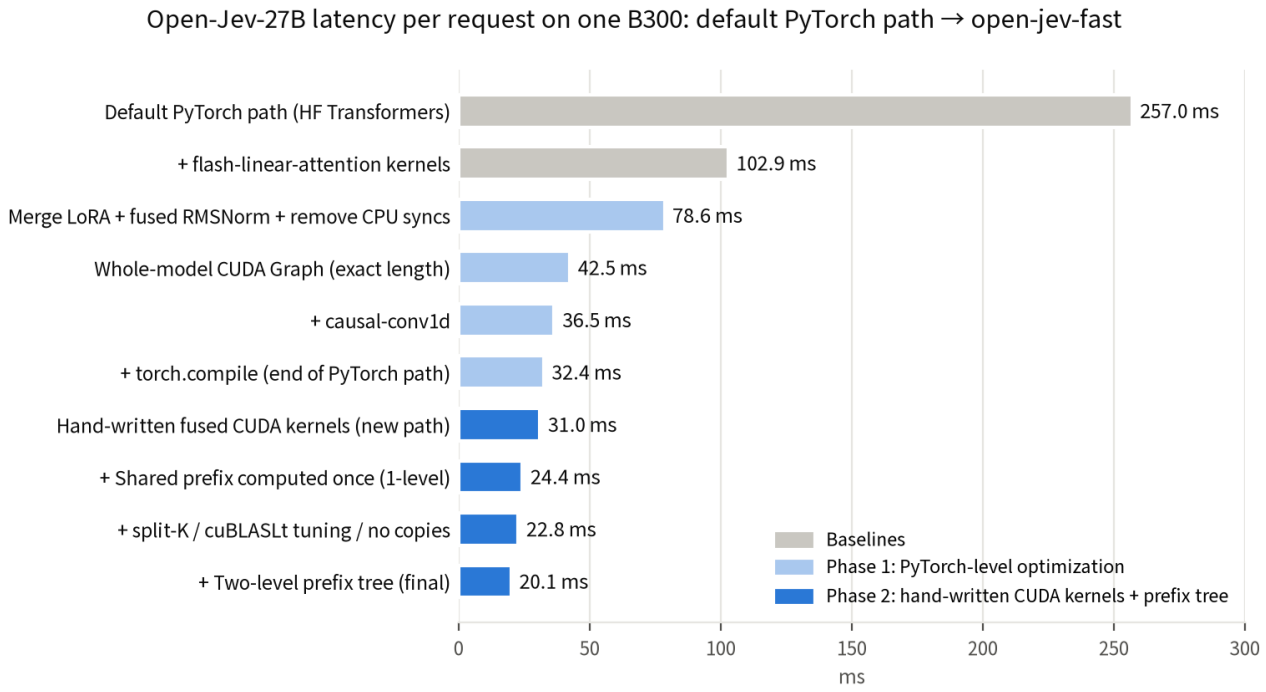

Figure 1. Latency of one inference on the example request as optimizations are added. Phase 1 works at the PyTorch level; phase 2 replaces the phase-1 path with hand-written CUDA kernels plus a prefix tree. Median latency of the forward pass plus scoring head.

1. Test conditions

Item	Single inference (Figure 1, 20.1 ms, etc.)	JevBench (Section 7)
Request	The repo's example configs/example-request.json: a customer-service conversation plus 3 questions (routing, 1 of 3; refund review needed, yes/no; urgency, 3 levels). 7 candidate prompts, 539 tokens in total	The 231 public JevBench [11] tasks (upstream f8ce713); each task has 1 question
Model	Open-Jev-27B-v1.1, bf16, not quantized	Same
Hardware	One dedicated NVIDIA B300 (sm_103)	Same; original and optimized both on a single B300. Both servers run with --max-length 16384 so every task fits (the model card's saved limit is 4,096 tokens per candidate)
What is timed	One full forward pass plus the scoring head, all 64 layers. **Everything is recomputed every time; nothing is reused across requests.** A shared prefix is computed once only within the same request. **Excludes** prompt building and tokenization (done once beforehand) and HTTP	Client-side: the per-task end-to-end latency recorded by the official TypeSafe adapter, including tokenization, HTTP and JSON. A new connection per task, concurrency 1

Item	Single inference (Figure 1, 20.1 ms, etc.)	JevBench (Section 7)
Statistic	5 warm-up runs, then the median of repeated runs, with cuda.synchronize before and after each run. The three headline numbers (257.0 / 102.9 / 20.1 ms) come from one script, 30 runs each. The intermediate steps are development measurements; 78.6 and 42.5 ms include tokenization (about 1–2 ms)	Service already warmed up (second pass; CUDA Graphs for the needed shapes already captured). The 231 tasks run once, in order
Correctness	Per-candidate probabilities within 0.0034 of the original (with FLA); all three decisions identical	Official score_task scoring; predictions compared task by task with the original

2. Where the time went

Open-Jev's default install has no FLA kernels, so Transformers runs the Gated DeltaNet layers as plain PyTorch ops: 257 ms. With FLA installed it takes about 103 ms. Measured there: **the moment the CPU finishes issuing all operations is the moment the GPU finishes computing.** Each request launches about 5000 GPU kernels, while the GPU is actually busy for only about 51 ms, so the bottleneck is the CPU issuing small operations one by one. The three main sources:

- the linear-attention operators (Triton) cost 65–150 µs of CPU time per launch, about 35 ms in total;
- LoRA was not merged, so each of the 160 projection layers had an extra fp32 branch: 320 extra matrix multiplies, about 20 ms;
- RMSNorm in transformers is split into 7 small operations, about 12.5 ms in total.

In addition, most of the content of the 7 candidate prompts is repeated (context and question text), and the original recomputes all of it for every candidate.

3. Phase 1: PyTorch level (103 → 32.4 ms)

Step	Single P50	Notes
Default PyTorch path (Open-Jev default install)	257.0 ms	Pure-PyTorch implementation of the linear-attention layers in HF Transformers [7]
+ flash-linear-attention kernels	102.9 ms	FLA [4] Triton kernels for the Gated DeltaNet layers
Merge LoRA [10] via PEFT [8], fuse RMSNorm, remove 2 CPU syncs	78.6 ms	Both syncs are in the transformers mask code, a "skip if there is no padding" shortcut; removing them does not change the math
Capture the whole model as one CUDA Graph	58.0 ms → 42.5 ms	With the syncs gone the whole model can be captured; capturing at the real length 82 instead of padding to 128
+ causal-conv1d kernel [6]	36.5 ms	Short convolution switched from the generic PyTorch implementation to the dedicated kernel
+ torch.compile [9] fusion, then capture	32.4 ms	Batched throughput 27 → 40.7 requests/s

JevBench with the same numerics: 198/231, identical to the original.

4. Phase 2: hand-written CUDA kernels + prefix tree (32.4 → 20.1 ms)

1. Hand-written fused kernels (31.0 ms). All non-matmul operations in every layer were replaced by 6 hand-written CUDA kernels: residual add + RMSNorm, SiLU×mul, linear-attention prep, gated RMSNorm, full-attention prep, and sigmoid-gate multiply. Linear-attention prep covers the causal convolution, SiLU, q/k/v split, head expansion, L2 normalization and gate parameters; full-attention prep covers QK normalization, partial RoPE and the layout change. Matrix multiplies were merged per layer, from 9 to 4. GPU kernels per request went from about 5000 to 945. Each kernel reproduces where the Transformers [7] and FLA [4] references round in bf16, and matches them bit for bit on 98–100% of elements on real activations. The Gated DeltaNet core [5] remains FLA's chunk_gated_delta_rule [4], and full attention is PyTorch SDPA.

2. Shared prefix computed once (24.4 ms). The prefix shared by all candidates and each candidate's tail go through one forward pass, so the weights are read once; computed rows 574 → 364. In the linear-attention layers

each candidate runs its recurrence over its own complete sequence, so the convolution window and recurrent state match the original; the full-attention layers use a tree mask (related: shared-prefix attention [12, 13] and tree attention [14, 15]). Long-context tasks benefit most: one task with a 3724-token shared context and 5 options went from 846 ms to 193 ms.

3. Matrix multiplies and data movement (22.8 ms).

- split-K for down and o_proj, with partial sums accumulated in fp32, which is more precise than a single matmul;
- measure every candidate cuBLASLt algorithm for each shape and keep the fastest; the full-attention input shape got 23% faster;
- removed the gather/cat copies used to replicate the prefix.

4. Two-level prefix tree (20.1 ms). The context is computed once, each question's text once, and each candidate only computes its tail; computed rows for the example request 364 → 277. Nodes are not padded to each other, and all structure lives in index tensors, so requests of the same size reuse the same CUDA Graph.

5. Why bf16 is hard to push further

Part (example request)	GPU time	Notes
Matrix multiplies	about 13 ms	Compute the 27B model cannot avoid. This card's measured bf16 peak is 1.76 PFLOPS; we already reach 70–80% of it
Linear-attention core (FLA)	about 3.7 ms	Tried FLA's token-by-token version; slower on the GPU (171 vs 77 µs per layer)
Own kernels, attention, gaps	about 3–4 ms	Another 1–2 ms could be fused away, but it is a lot of work
FP8 matrix multiplies (measured, not used)	13 → about 7 ms	Would bring one inference to an estimated 12–13 ms, but changes the numerics; accuracy must be re-validated

6. Correctness

Checked at three levels: 1) each kernel compared element by element with the PyTorch reference on real activations; 2) hidden states compared layer by layer over all 64 layers, with a 0.19% relative error on the last token read by the scoring head; 3) all 231 JevBench tasks end to end, with the official TypeSafe adapter and scoring code, predictions compared task by task.

Version	Accuracy	Tasks that changed vs. original
Original	198/231	—
PyTorch-optimized	198	1 task (wrong before and after)
Service v1 (1-level prefix tree + graphs)	198	1 task (wrong before and after)
Service v3 (2-level prefix tree + tuning)	197	2 tasks: 1 wrong before and after, 1 coin flip

The coin-flip task is hard-opus-a-temporal_numeric-09: original no/yes = 0.504/0.496, merged LoRA 0.503/0.497, hand-written kernels (without the prefix tree) 0.496/0.504. Any rounding difference under 0.01 flips it; it is unrelated to the prefix tree. Probability differences grow with input length, because bf16 rounding-order differences accumulate through 64 layers and the recurrent state: at most 0.0034 on the example request (the two original paths, with and without FLA, differ from each other by 0.0087), and up to 0.035 on the longest request tested (10,722 tokens), with the decision unchanged. The model card reports 197/231 on public JevBench for this checkpoint; our run of the original server scored 198/231.

7. JevBench measurements (HTTP, concurrency 1, warmed up)

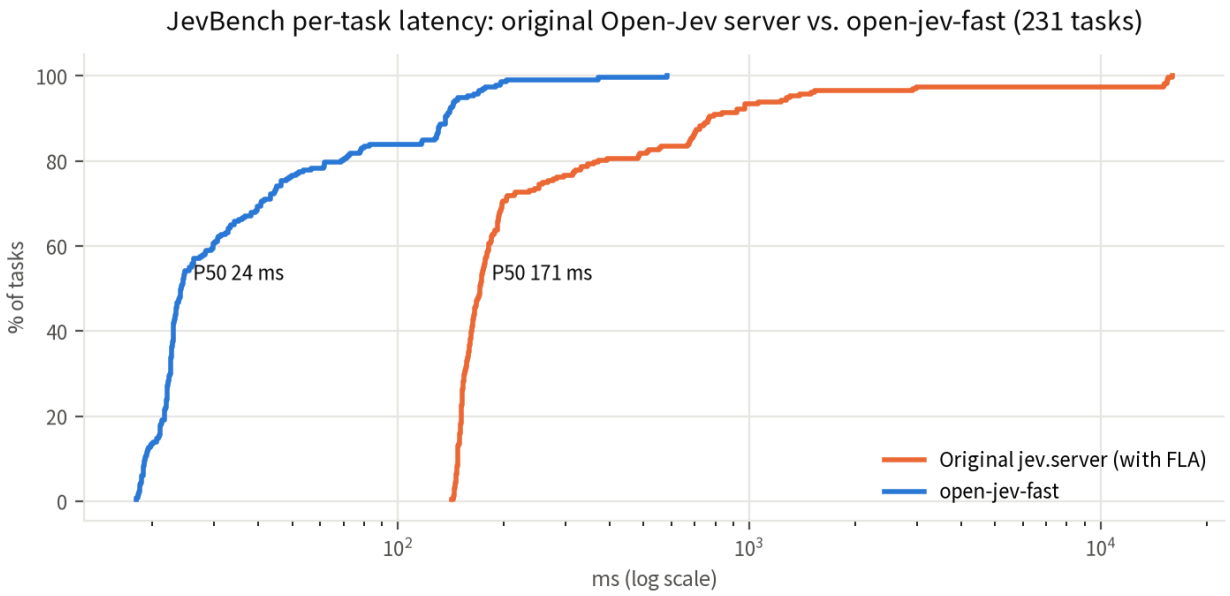

Figure 2. Cumulative share of the 231 JevBench tasks finished within a given latency (ms, log scale), sent over HTTP to the local server, one request at a time.

	Original jev.server (with FLA)	Service v1	Service v3 (current)
Accuracy	198/231	198/231	197/231
Mean	703 ms	52.1 ms	51.3 ms
P50	171 ms	28.6 ms	24.2 ms
P95	1272 ms	145 ms	148 ms
Max	16018 ms	492 ms	583 ms
Total for 231 tasks	162 s	12 s	about 12 s

The v3 server-side inference itself has a P50 of 18.6 ms; the rest is HTTP. The JevBench client opens a new connection per task, which adds about 25 ms; with a reused connection (keep-alive) the HTTP overhead is about 4 ms. Each JevBench task has only one question, so the two-level prefix tree does not help here; it mainly speeds up multi-question requests.

8. Failed or abandoned attempts

Attempt	Result
The repo's built-in prefix cache score_cached	1205 ms, 10× slower: split into 10 calls, too much CPU overhead
torch.compile default mode; CUDA Graph mode before removing the syncs	No gain / only 10% faster, because the graph gets broken
FLA token-by-token version instead of the chunked one	Slower on the GPU (171 vs 77 µs per layer)
split-K with bf16 partial sums	Faster but noisier; switched to fp32
Swapped matmul layout; padding rows to a multiple of 128	Gain ⩽ 0.2 ms; dropped
v2 service using the replicated layout on long prefixes too	Long tasks 30–60% slower; v3 switched to a hybrid strategy
Writing the access log directly to the network disk on every request	+9 ms per request; switched to an in-memory buffer flushed every 5 s

9. Scope and limitations

- Specialized for one model (Open-Jev-27B-v1.1 / Qwen3.8-27B shapes);
- **tested only on an NVIDIA B300**; the kernels are plain CUDA C++ compiled for sm_103 by default, and another GPU with enough memory needs a different -gencode flag in src/ext.py (untested; speedups will differ);
- **GPU memory: about 96 GiB** (98,280 MiB in nvidia-smi after the full JevBench run), so an 80 GB GPU is not enough as is. This covers the weights, the merged and split weight copies (the originals are kept, so a large part is duplicated), the CUDA Graph pool and activations;
- bf16 only; FP8 matrix multiplies were measured but not used, because they change the numerics;
- requests are processed one at a time;
- the first request of a new shape captures a CUDA Graph, adding 1–3 s to that request; clients should reuse connections (keep-alive).

10. References

[1] Open-Jev contributors. Open-Jev. github.com/Zefan-Cai/Open-Jev (commit 3308a15), 2026. MIT.

[2] Open-Jev-27B-v1.1. huggingface.co/ZefanCai/Open-Jev-27B-v1.1. Apache-2.0.

[3] Qwen Team, Alibaba Cloud. Qwen3.8-27B. huggingface.co/Qwen/Qwen3.8-27B (revision 1d4bf0f). Apache-2.0.

[4] S. Yang and Y. Zhang. FLA: A Triton-Based Library for Hardware-Efficient Implementations of Linear Attention Mechanism. github.com/fla-org/flash-linear-attention, 2024.

[5] S. Yang, J. Kautz and A. Hatamizadeh. Gated Delta Networks: Improving Mamba2 with Delta Rule. ICLR 2025. arxiv.org/abs/2412.06464; S. Yang et al. Parallelizing Linear Transformers with the Delta Rule over Sequence Length. NeurIPS 2024. arxiv.org/abs/2406.06484

[6] T. Dao. causal-conv1d. github.com/Dao-AILab/causal-conv1d. BSD-3-Clause.

[7] T. Wolf et al. Transformers: State-of-the-Art Natural Language Processing. EMNLP 2020 System Demonstrations. aclanthology.org/2020.emnlp-demos.6

[8] S. Mangrulkar et al. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods. github.com/huggingface/peft, 2022.

[9] J. Ansel et al. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. ASPLOS 2024.

[10] E. J. Hu et al. LoRA: Low-Rank Adaptation of Large Language Models. ICLR 2022. arxiv.org/abs/2106.09685

[11] F. Standhartinger and contributors. JevBench. github.com/fstandhartinger/jevbench (commit f8ce713), 2026. MIT.

[12] J. Juravsky et al. Hydragen: High-Throughput LLM Inference with Shared Prefixes. 2024. arxiv.org/abs/2402.05099

[13] L. Zheng et al. SGLang: Efficient Execution of Structured Language Model Programs. NeurIPS 2024. arxiv.org/abs/2312.07104

[14] X. Miao et al. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. ASPLOS 2024. arxiv.org/abs/2305.09781

[15] T. Cai et al. Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads. ICML 2024. arxiv.org/abs/2401.10774

[16] B. Zhang and R. Sennrich. Root Mean Square Layer Normalization. NeurIPS 2019. arxiv.org/abs/1910.07467

[17] J. Su et al. RoFormer: Enhanced Transformer with Rotary Position Embedding. 2021. arxiv.org/abs/2104.09864

[18] NVIDIA. cuBLAS/cuBLASLt documentation (docs.nvidia.com/cuda/cublas); CUTLASS (split-K GEMM), github.com/NVIDIA/cutlass.

What is used from each project is listed in THIRD_PARTY_NOTICES.md in the repository.