Inference · Technical report

Fitting a 27B model, its drafter, and a conversation into 12 GB

Trevor Rosato

Speculative decoding makes large models fast, but it needs memory a small card does not have. Measuring what verification actually costs, and replaying accepted work instead of snapshotting it, cut the memory speculation held in reserve by 41 times and brought a 27-billion-parameter model to 148 tokens per second on an RTX 4070.

November 5, 2026 Work conducted September 2026 10 minute read

This report describes one campaign from our consumer-GPU research: serving a compressed dense 27-billion-parameter model, Qwen3.8-27B in EschaLabs' two-bit Escha-W2 compression,[1] on an RTX 4070 with 12 GB of memory, fast enough for interactive use and with room for an actual conversation.

The weights themselves fit. The hard part was everything else that has to share the card with them. We use speculative decoding: a small draft model proposes several tokens ahead, and the large target model checks them all in one pass, keeping the ones it agrees with.[2] Done well, the target emits several tokens for the cost of one forward pass. But the drafter needs memory, and so does the bookkeeping that lets the target undo the proposals it rejects. On a 12 GB card, that bookkeeping turned out to be the largest obstacle.

The best configuration reached a warm median of **148.03 tokens per second** on short requests. Getting there took four separate findings.

Eight projection rows cost about as much as one

Speculative decoding only pays off if checking several proposed tokens is cheap compared to generating them one at a time. Published curves for that trade-off come from other models, other quantizations and other runtimes, so we measured it for this target directly. We captured the target's 368 projection operations and timed them as the number of positions being verified grew.

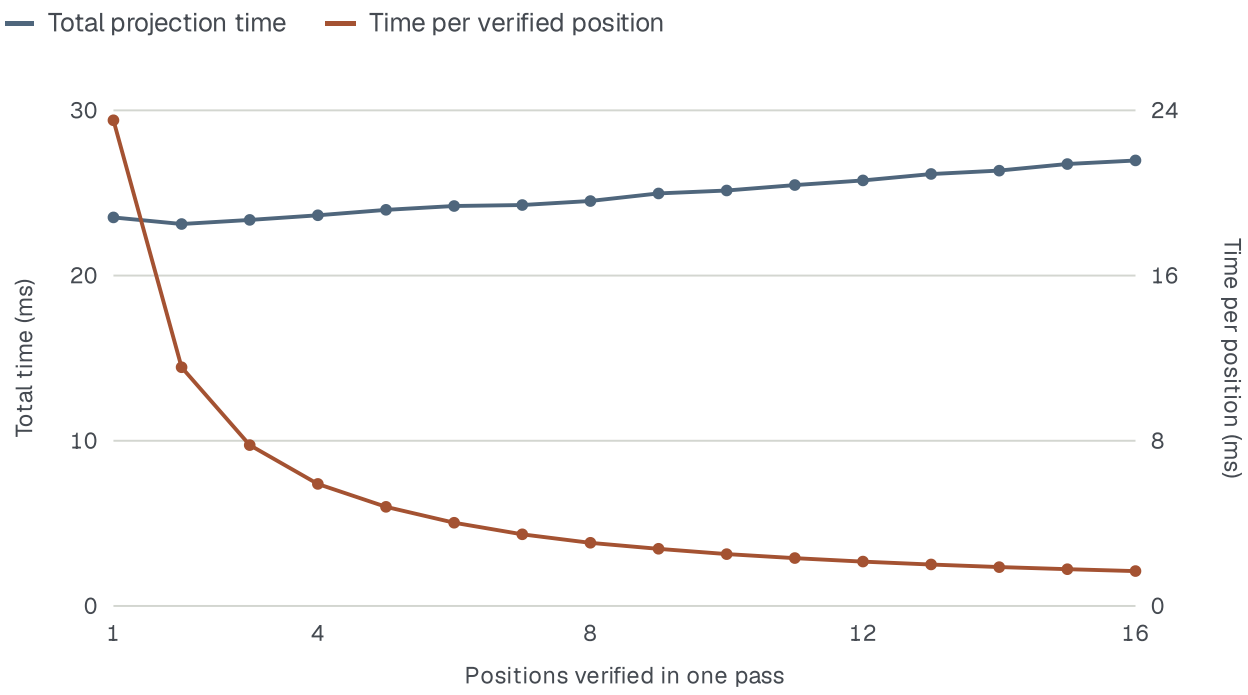


Figure 1
The target's projection workload, timed with CUDA events. Eight positions take 24.5 ms against 23.5 ms for one, only 4% more, so the projection cost per position falls by almost eight times. The flat curve is what makes wide speculation worth pursuing on this card.

The curve is nearly flat. Eight positions cost about 4% more than one, so the projection work can be spread over many more accepted tokens. That made the next question urgent: if the target can afford to check many tokens, can the rest of the system afford to propose them?

Replay accepted work instead of snapshotting it

This target is a hybrid. Some of its layers are recurrent: instead of attending over the whole history, they carry a compact running state forward one token at a time. Speculation complicates that. While the target is checking eight proposed tokens, it does not yet know how many it will accept, so it has to be able to return to the state after any of them.

The straightforward approach keeps a full copy of the recurrent state for every proposed position. At eight positions, that reserved **1,174.5 MiB**, about a tenth of the card, before the drafter or a single conversation had any memory at all.

We ported the approach from ReplaySSM:[3] keep only the inputs needed to recompute the state updates, and when the target decides how many tokens it accepts, replay just those updates forward from the last committed state. Rejected proposals are simply discarded.

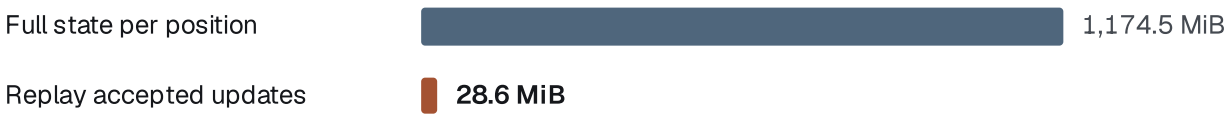


Figure 2
Persistent memory reserved for speculation at eight positions, for one request. Replaying accepted updates instead of snapshotting every position cut it by 41 times and returned more than a gigabyte to the card.

The reduction was **41 times**, and the 1.12 GiB it freed is memory that conversations can use instead. Nothing about the target's weights changed.

The detail that made it work was numerical. Replaying an update is only correct if it reproduces the original exactly, and a different operation order or precision is enough to drift the state, since floating-point results depend on both.[4] We reused the production recurrence kernel for the replay rather than writing a new one, and batched independent layers through it. One bug is worth passing on to anyone doing similar work: writing into a ring buffer through advanced indexing followed by a copy can update a temporary tensor instead of the buffer itself, silently.

A drafter needs the computation it was trained with

Freeing memory for a drafter only helps if the drafter proposes tokens the target accepts. Our first integration used a DFlash2 drafter[5] in a different numeric format from the target: both use sixteen bits, but one is FP16 and the other BF16, which trade precision for range differently. Running the drafter in the target's format produced overflows and garbage proposals. Giving it its own precision, with explicit conversions where the two models share embeddings and features, fixed that.

Precision was not the only mismatch. A small checkpoint does not describe the full computation it was trained to perform; the drafter's feature processing, convolution and attention had to be ported exactly as trained, not approximated with a generic backbone.[6] With both in place, the target emitted an average of **6.86 tokens per verification pass** on a code-generation prompt, and the integration ran at **100 tokens per second**. A later drafter, Apathy v3, produced the 148 tokens-per-second configuration.

The workload sets the speed

That 6.86 tokens per pass came from asking for a Fibonacci function, a highly predictable piece of code. Across a broader set of prompts, the picture changed.

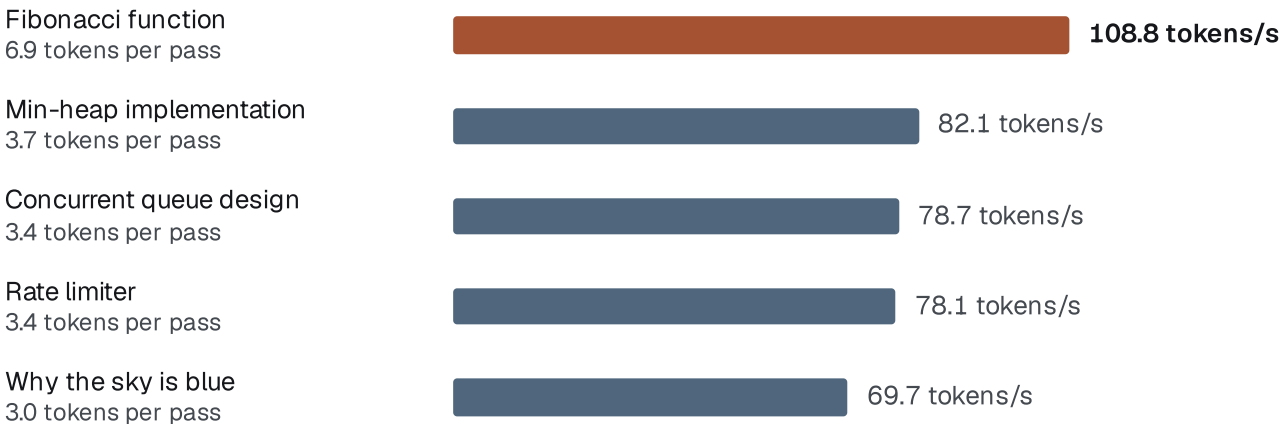


Figure 3
The same configuration on five prompts. Throughput follows how many proposals the target accepts, which follows how predictable the text is. Prose explaining physics is harder to guess than boilerplate code.

Speed in speculative decoding is a property of the configuration and the workload together, because the gain depends on how often the target accepts the draft.[2] A horizon that looks ideal on a predictable benchmark spends time drafting and reserving memory for tokens the target rejects on harder text. For a tutor, whose output is mostly explanation rather than code, the honest number is closer to the bottom of this chart than the top. We now track acceptance per workload alongside throughput, and choose speculation settings for the text we actually serve.

Kernels shaped for verification

The last finding concerned attention during verification. The attention kernel for compressed, four-bit caches was written for the common case of generating one token at a time. Verification checks several positions at once over a long history, and the single-token kernel left most of the GPU idle.

We wrote an experimental kernel that splits the long history into parts processed in parallel, reuses small tiles of queries, and merges the partial results. These are established techniques, used for the same reason in Flash-Decoding;[7] the work was applying them to this cache format and these shapes. On a synthetic case with eight queries over a 12,288-token history, attention time fell from **5.6 ms to 0.97 ms**, 5.8 times faster, and a nineteen-query case ran 2.75 times faster. This kernel is not yet part of the end-to-end numbers above.

What this campaign changed

The largest gain came from neither the model nor the hardware. It came from how the runtime represented speculative work, and the model's file size would never have pointed us there. Fitting a capable model on a small card is mostly a memory accounting problem: weights, drafter, recurrent state, conversation caches and scratch space all compete for the same 12 GB, and the biggest wins come from finding the item on that list that is far larger than it needs to be.

Methods and scope

All measurements come from one RTX 4070 (12 GB) in a Windows/WSL host with a Ryzen 7 7700X and 32 GB of system memory. Serving rates are warm medians over short prompts with 96 generated tokens, computed from local HTTP request wall time. The two drafter configurations are different checkpoints and are reported separately. The projection curve,

speculative-storage comparison and attention kernel are component measurements; the workload sweep is one observation per prompt. These results describe serving speed, not answer quality, which we evaluate separately.

The target model and compression come from Qwen and EschaLabs; ReplaySSM and DFlash are upstream research, and syvai supplied the DFlash2 checkpoint variant. The runtime builds on SGLang.[8] Our contribution was the serving integration, the state and execution changes, and the measurements.

References

1. EschaLabs, Qwen3.8-27B-Escha-W2 https://huggingface.co/EschaLabs/Qwen3.8-27B-Escha-W2.
2. Yaniv Leviathan, Matan Kalman and Yossi Matias, “Fast Inference from Transformers via Speculative Decoding” https://proceedings.mlr.press/v202/leviathan23a.html, ICML 2023.
3. Ze-Wei Liou et al., “ReplaySSM: Cache SSM Inputs, Not State” https://arxiv.org/abs/2610.13778, 2026; code at ReplaySSM https://github.com/Johnny-Liou/ReplaySSM.
4. PyTorch, “Numerical accuracy” https://docs.pytorch.org/docs/2.14/notes/numerical_accuracy.html, PyTorch 2.14 documentation.
5. DFlash https://github.com/z-lab/dflash; syvai, Qwen3.8-27B-DFlash2-W4A16 https://huggingface.co/syvai/Qwen3.8-27B-DFlash2-W4A16.
6. Inco AI, “DFlash 2: Keep Drafting Parallel” https://inco.ai/blog/dflash2/, 2026.
7. Tri Dao, Daniel Haziza, Francisco Massa and Grigory Sizov, “Flash-Decoding for long-context inference” https://pytorch.org/blog/flash-decoding/, PyTorch Blog, 2023.
8. SGLang https://github.com/sgl-project/sglang.

Cite this

Trevor Rosato. "Fitting a 27B model, its drafter, and a conversation into 12 GB." research.honorlyai.com, November 5, 2026. https://doi.org/10.5281/zenodo.24137187