

SR-HPM: Speculative Verification on the CPU

Fast decoding of dense LLMs larger than GPU memory on consumer hardware

Javitax47*

October 2026

Abstract

A dense 27B-parameter model quantized to ~ 3.5 bits does not fit in the 6 GB of a laptop GPU. Partial offload leaves most weights in system RAM, and decoding becomes bound by reading them once per generated token. Speculative decoding should amortize that read over several tokens, but on the CPU path of llama.cpp verifying M drafted tokens costs $2\text{--}4\times$ a single token, so drafters barely help. We present SR-HPM, a patch to llama.cpp and a launcher that make verification on the CPU nearly memory-flat (4 tokens for $1.1\text{--}1.2\times$ the cost of one) and then remove the remaining costs of the pipeline one measurement at a time: cost-aware layer placement, a lossless 4-bit re-encoding of grid-quantized weights, lazy rollback snapshots for hybrid recurrent models, a split prompt-processing path, co-verification by the otherwise idle GPU reading weights from RAM over PCIe, large pages, and kernel-aware re-quantization. On Qwen3.8-27B with an RTX 3060 Laptop and a Ryzen 7 5800H, greedy decoding goes from 2.7 tok/s (stock llama.cpp with the same drafter) to 9.6–10.2 tok/s with identical output, and ~ 10.4 tok/s with an opt-in relaxed acceptance that costs 0.018 nats per token. We also report about twenty ideas that did *not* work—draft trees, rerankers, activation sparsity, layer dropping, 2-bit layers with low-rank correction among them—each with the cheap experiment that refuted it, and identify the two walls that remain: RAM bandwidth in the typical 4-token pass, and the drafter’s acceptance rate.

1 Introduction

Running a model whose weights exceed GPU memory is the common case on consumer machines. llama.cpp [3] splits such a model by layers between GPU and CPU; for dense models the CPU part dominates, and one decoding step reads every CPU-resident weight from DRAM. On our reference laptop (dual-channel DDR4-3200, ~ 38 GB/s measured) the ~ 9 GB that do not fit in VRAM bound decoding to a few tokens per second.

Speculative decoding [1, 2, 6] turns this around in principle: a cheap drafter proposes several tokens and the target model verifies them in one pass, so one read of the weights yields several tokens. The premise is that verifying M tokens costs about as much as one, which holds on GPUs. It does not hold on the CPU path of llama.cpp: its quantized matrix kernels re-decode each weight block for every token, so a 4-token pass costs $2.1\text{--}2.7\times$ a 1-token pass (Table 1). With a strong drafter (DFlash2) stock llama.cpp reaches 2.7 tok/s on Qwen3.8-27B.

SR-HPM starts from a verification kernel that decodes each block once for all tokens, and then follows the cost of a decoding step wherever it goes. Our contributions are:

- small-batch CPU kernels for the quantization formats of modern dynamic quant, including a lossless 4-bit re-encoding (C4) that replaces grid lookups by a single shuffle, sign-free paths for offset-coded types, and a paired layout for IQ4_XS (§3);
- rollback machinery for hybrid recurrent models (gated delta networks) that keeps two full states per pass and replays the rest exactly from rank-1 records, and a split prompt-processing path that keeps those snapshots exact at GPU speed (§4);
- co-verification by the discrete GPU, which reads a share of the CPU layers’ weights directly from pinned RAM over PCIe while the CPU computes the rest (§5);
- placement, drafting, quantization and system choices, each justified by an A/B measurement, and a launcher that applies them to any dense GGUF model (§6);
- a catalogue of refuted ideas with the experiments that refuted them (§8), and an analysis of where the remaining time goes.

*Code, data and lab notes: <https://github.com/Javitax47/sr-hpm>. The experiments, code and text were produced with the assistance of an AI coding agent (Claude, Anthropic) under the author’s direction.

M (tokens)	our kernel (ms)	cost vs $M=1$	llama.cpp Q3_K	llama.cpp Q4_0
1	1.62	1.00	1.00	1.00
2	1.75	1.08	1.53	1.28
4	1.92	1.18	2.73	2.05
8	2.87	1.77	3.89	3.30

Table 1: Cost of one 17408×5120 4-bit matrix product (does not fit in L3) for M tokens, 8 threads. The first kernel prototype already makes 4 tokens cost $1.18\times$ one; llama.cpp’s CPU path costs $2\text{--}2.7\times$.

2 Setting and methodology

Hardware and model. Laptop with an RTX 3060 Laptop GPU (6 GB, PCIe 3.0 x8), Ryzen 7 5800H (8 Zen 3 cores, AVX2, no VNNI), 32 GB DDR4-3200, Windows 11. Target: Qwen3.8-27B, a hybrid model with 64 layers (48 gated-delta-net recurrent layers, 16 full-attention layers), quantized from Unsloth’s UD-Q3_K_XL dynamic recipe (12.2 GB) and later from our own kernel-aware mix (12.4 GB). Drafter: DFlash2 (block 8, a 16-candidate selector lattice), Q4_K_M, on the GPU.

Metrics. Decode speed is the mean over 10 fixed prompts (code, math, prose, translation; Spanish and English), 200 greedy tokens each, with thinking disabled. A hash of the outputs certifies exactness: changes that must not alter the output are rejected if the hash changes. Quality is the KL divergence to the Q8_0 model on 16×512 tokens of a held-out corpus (`llama-perplexity -kl-divergence`), with the fraction of positions where the top token agrees. τ is the number of tokens produced per verification pass.

Noise. The laptop drifts by $\pm 10\%$ with temperature and background load. Every comparison is made in pairs within one session (A B A B); we report both runs. Differences inside the pair spread are reported as ties.

3 Small-batch verification kernels

The kernel family `GGML_SRV` decodes each 256-weight super-block once into signed 8-bit values and applies it to the $M \leq 8$ activation rows held in registers or L1 (`vpmaddubsw` on $|w|$ and sign-adjusted activations, then `vpmaddwd` and 32-bit accumulation). Activations are quantized per row to 8 bits with per-block scales, bit-identically to ggml’s Q8_K, so the result equals stock llama.cpp.

C4: lossless 4-bit re-encoding of grid quants. IQ3_S and IQ3_XXS store indices into a 512- or 256-entry grid of 4 values each; decoding needs one gather per 4 weights, which dominates once the data is in cache. Within a 32-weight sub-block those grids use at most 16 distinct magnitudes, so we re-encode the weights losslessly as 4-bit codes plus a 16-entry table and decode with one `pshufb`: $1.5\text{--}1.8\times$ faster, 4.375 bits per weight.

Sign-free paths. Q3_K, C4 and Q6_K store $w + o$ with a constant offset. Multiplying the unsigned code directly and subtracting $o \cdot \sum x$ per group removes the per-token sign step (one instruction of five); the group sums come from one vector per super-block.

Paired IQ4_XS. Loading two 32-weight sub-blocks per 32-byte load and permuting the activations once per pass instead of shuffling weights per block cuts IQ4_XS (53% of the CPU bytes) by 5–7%.

The wall. With these kernels the CPU reads IQ4_XS at 35 GB/s for 4 tokens, against a measured RAM ceiling of 38–41 GB/s; with 8 tokens it is compute-bound at ~ 20 GB/s. A synthetic peak benchmark puts the Zen 3 compute ceiling at 1.48 cycles per 32 weights and token with the sign step and 1.10 without, against 2.0–2.3 in the real kernel.

4 Rollback for hybrid recurrent models

Speculative decoding must undo rejected tokens. For attention layers this means truncating the KV cache; for recurrent layers llama.cpp keeps $K = n_{\text{draft}} + 1$ snapshots of the state per layer (a 128×128 matrix per head, 3 MB per layer) and writes all of them every pass—400–800 MB of writes per pass on Qwen3.8-27B.

Direct and lazy snapshots. We first write the snapshots straight into the cache from the CPU kernel instead of through the op output and a copy (+6%). Then we observe that the gated delta rule $S_t = g_t S_{t-1} + k_t \delta_t^\top$ is a rank-1 update: keeping two full states (the newest and the oldest slot) and the vectors (k_t, δ_t, g_t) of the tokens in between (~ 1 KB per head) allows an exact replay of any intermediate state when a rollback actually happens. The output hash is unchanged and decoding is 4% faster.

Split prompt processing. With rollback enabled, prompt processing was 35% slower than without a drafter. Removing the drafter’s injection or feature extraction changed nothing; forcing rollback snapshots without a drafter reproduced the loss exactly. Two independent costs: (i) 8 strided copies per layer of the convolution state out of the whole ubatch (~ 5 s per 3.6k-token prompt), replaced by one gather of the last $K + 2$ columns; (ii) the lazy recurrent op exists only on the CPU, where it runs token by token, while the regular op is offloaded to the GPU for large batches and runs in chunks. For prompt ubatches the lazy op now processes only the first token (rebuilding the state of a pending rollback) and the regular op the rest, writing full snapshots. Prompt processing goes from 98 to 132 tok/s (145 without a drafter), with identical output.

5 dGPU co-verification over PCIe

During the CPU part of a pass—most of it—the GPU is idle. We let it compute the first rows of every IQ4_XS and Q3_K product of the CPU layers, reading a pinned copy of those rows from RAM over PCIe with zero-copy loads (one warp per row streams the row through shared memory with consecutive 4-byte loads), while the CPU threads compute the remaining rows. Measured alone, PCIe 3.0 x8 delivers 6.7 GB/s, and with the CPU kernel running concurrently the two add up to 29.3 GB/s without slowing the CPU. In the model the gain depends on M : with 4 tokens the CPU already reads 35 GB/s, and PCIe reads only compete for the same DRAM (890 $\rightarrow$ 1120 μ s per product); from 6 tokens the CPU is compute-bound and the GPU’s rows are a net gain (-15% per product at $M = 8$). With a per- M share table (0 up to 5 tokens, then 12%, 16%, 20%) the pass is 2–4% shorter. The GPU rows use float activations and are slightly more precise than the CPU’s 8-bit path.

6 System-level choices

Placement. VRAM buys CPU time, not bytes: we rank layers by measured CPU seconds per byte of their tensor types and put the best contiguous range on the GPU (every CPU $\leftrightarrow$ GPU boundary costs a synchronization), plus the output head when the drafter shares it. llama.cpp’s op offload sent every CPU layer’s normalization to the GPU (64 boundaries per token); offloading only large-batch products restores 4 boundaries.

Adaptive drafting. Because 4-token verification is nearly free and 8-token verification is not, we always keep 3 drafted tokens and extend up to 7 only while the drafter’s selector probability stays ≥ 0.9 (τ 3.25 $\rightarrow$ 4.2, +7–10%). Choosing the length by expected value from the measured cost curve ties.

GPU keep-alive and large pages. Between bursts the driver drops the GPU to a low clock state; a 1-warp spin kernel on a separate stream keeps it up (+0–5%, drafter 19 $\rightarrow$ 11 ms). With the “Lock pages in memory” right, 8 GB of CPU weights live on 2 MB pages, which removes most TLB walks of the 8.8 GB read per pass. Measured in two pairs: +10% (9.24 vs 8.40 tok/s) and a tie (8.37 vs 8.47), with identical output—a gain of 0–10% that the noise of this session could not pin down.

Kernel-aware quantization. The CPU path reads IQ3 through C4 at 4.375 bpw—more bytes than IQ4_XS (4.25) for less quality—while Q3_K is read natively at 3.44 bpw. Starting from the dynamic recipe, rewriting only the CPU layers’ IQ3 tensors gives two operating points: *quality* (IQ3 $\rightarrow$ IQ4_XS: KLD 0.0427 $\rightarrow$ 0.0345 at the same speed) and *speed* (FFN IQ3 $\rightarrow$ Q3_K, attention IQ3 $\rightarrow$ IQ4_XS: same KLD, -6.6% CPU bytes, +5.7%).

Relaxed acceptance (opt-in). At the position where a draft breaks, the drafted token is often the target’s second choice with a similar probability. Accepting a drafted token whose target logit is within δ nats of the best one yields +6% at $\delta = 1$ for 0.018 nats per token given up—less than half of what the quantization already costs (Table 2).

δ	tok/s	τ	acceptance	cost (nats/token)
0 (exact)	9.80	4.08	0.728	0
0.5	9.85	4.13	0.756	0.004
1	10.40	4.45	0.807	0.018
2	10.70	4.67	0.841	0.053
3	10.84	4.66	0.844	0.075

Table 2: Relaxed acceptance. The cost is the log-probability given up against the target’s argmax; for scale, the ~ 3.5 -bit quantization costs 0.043 (KLD to Q8_0).

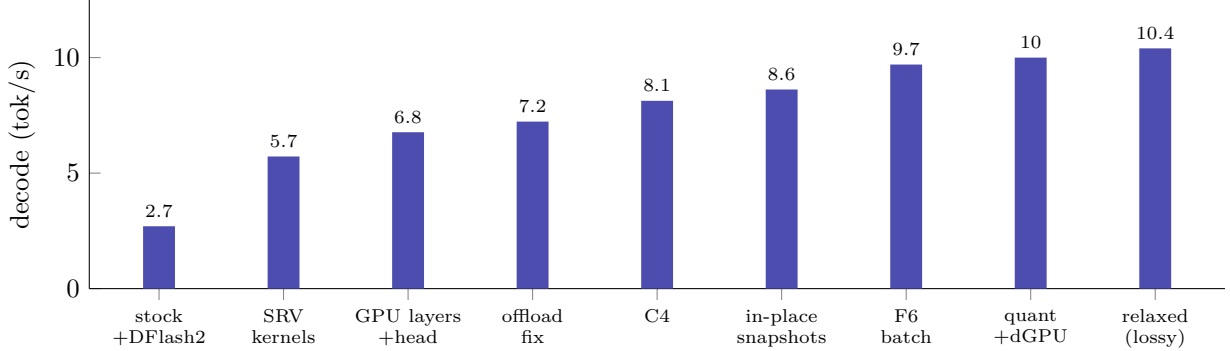


Figure 1: Decode speed of Qwen3.8-27B on the reference laptop as the components are added (10 prompts for F6 steps, 5 for the earlier ones). All configurations up to “relaxed” produce the target’s greedy output.

7 Results

Figure 1 shows the progression from stock llama.cpp (2.7 tok/s) to 9.6–10.2 tok/s with identical output. The largest steps are the small-batch kernels (2.7→5.7), C4 and in-place snapshots (7.2→8.6), adaptive drafting and the F6 batch of kernel, rollback and system changes (8.6→9.7), and kernel-aware quantization and dGPU co-verification (to ~ 10).

Where the time goes. A per-op profile of a pass (~ 0.45 s) attributes 76% to the CPU matrix products (IQ4_XS 55%, Q3_K 27%), 10% to other CPU ops (recurrent op 4%), and 14% to the GPU layers, the drafter (11 ms), feature injection (2 ms) and fixed overheads. The pass time grows from 0.34 s at $M = 4$ to 0.51 s at $M = 8$ (Figure 2).

Long contexts and prose. The pass time does not grow with context up to 4.8k tokens. On summaries of Spanish prose the drafter’s acceptance falls to 0.45–0.58 and decoding to ~ 6 tok/s on the same hardware: SR-HPM’s gains scale with the drafter’s quality.

Generality. Nothing in the kernels or the launcher is specific to Qwen: the launcher reads any GGUF, chooses between all-GPU, MoE and SR-HPM paths, and places layers by the measured cost of their tensor types. MoE models do not benefit—each verified token activates different experts, so verification is not memory-flat.

8 What did not work

Table 3 lists ideas that failed, including training-free activation sparsity [4] and layer dropping [5]. Three findings shape what is left. First, with 4 tokens per pass the CPU already reads at $\sim 90\%$ of the measured DRAM ceiling: any further gain at that point must reduce bytes, and every training-free way of doing so we tried (fewer bits, sparsity, dropping layers, a pruned head) costs more quality than it saves. Second, the drafting *policy* is exhausted—an oracle that stops each chain exactly at its first error gains only 9%—but the drafter’s 16 candidates contain the right token at 15–19% of the positions where its first choice is wrong, so a perfect selection would raise τ from 4.1 to 6.3. Telling those candidates apart without training costs as much as verifying them: the GPU-resident first layers do not know (10.9%), and trees pay for the extra rows. Third, quantization quality is not predicted by weight error alone: products that write into the residual stream are $\sim 16\times$ more sensitive per unit of weighted error.

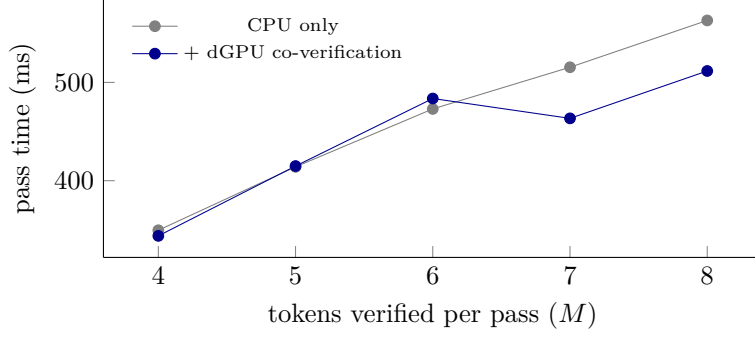


Figure 2: Wall time of a verification pass versus tokens verified, measured inside the server during decoding (SRHPM_PASS_LOG), with and without dGPU co-verification.

Idea	Refuting measurement
Lossless entropy coding of the weights	IQ4_XS codes carry 3.91 of 4 bits ($\leq 3\%$)
Delta verification $W(x_t - x_0)$	deltas are 50–95% of the activation norm
SMT helper threads prefetching weights	the kernel is compute-bound with ≥ 4 tokens
iGPU as second verifier (OpenCL)	38 GB/s alone; 5.5 $\times$ slower with the CPU busy (APU power budget)
Draft trees / two chains	$\leq +3\%$ with the real cost curve; on a recurrent model chains duplicate the prefix
Expected-acceptance path over the lattice	+0.3% τ
Online reranker of the 16 candidates	τ 4.098 vs 4.105 (a perfect one: 6.33)
Logit lens of the GPU layers as reranker	layer 20 picks the right candidate 10.9% of the time
2-bit CPU layers	KLD 0.094 and <i>slower</i> (7.8 vs 9.6 tok/s): grid decode is compute-bound
Low-rank GPU correction of 2-bit error	residual nearly white: rank 256 removes 10–20%
Sensitivity-guided mix (imatrix-weighted error)	predicted KLD 0.042, measured 0.057; fitted: -1.8% bytes
Pass-shared activation sparsity	skipping 10% of blocks costs 5–10 $\times$ the quantization error
Layer dropping	2 layers: KLD 0.043 $\rightarrow$ 0.108
Exact output head on a vocabulary subset	the bound adds 60–95 logits; 97–100% of the head is read
Small ops on one thread, fewer barriers	same speed; cache misses, not barriers
N-gram retrieval drafting	never fires on chat or summaries (paraphrase)
More GPU layers in spare VRAM	below ~ 0.4 GB free, WDDM slows everything down

Table 3: Refuted ideas. Scripts and data in [research/](#).

9 Limitations and future work

The fast kernels need x86-64 with AVX2; VNNI and AMX machines and ARM are untested. Rollback assumes one sequence. The dGPU co-verifier is implemented for Windows and CUDA. All measurements are on one laptop; desktop CPUs with DDR5 and more VRAM shift the balance (more layers on the GPU, more bandwidth), and Linux frees ~ 1 GB of VRAM that WDDM reserves. The two levers left with large margins require training, which is outside the scope of a tool for existing models: adapting the drafter’s candidate selector to the quantized target (the 4.1 $\rightarrow$ 6.3 headroom), and layer dropping with a short healing fine-tune.

10 Conclusion

Speculative decoding helps memory-bound CPU inference only if verification is memory-flat. Making it so on a consumer CPU, and then removing the rollback, prompt-processing, placement and idle-GPU costs that appear next, turns a 27B dense model on a 6 GB laptop GPU from 2.7 into ~ 10 tokens per second with identical output. The remaining distance to 20 is set by DRAM bandwidth and by the drafter, not by the software around them.

References

- [1] Y. Leviathan, M. Kalman, Y. Matias. Fast inference from transformers via speculative decoding. ICML 2023.

- [2] C. Chen et al. Accelerating large language model decoding with speculative sampling. arXiv:2302.01318, 2023.
- [3] G. Gerganov et al. llama.cpp. <https://github.com/ggml-org/llama.cpp>.
- [4] J. Liu et al. Training-free activation sparsity in large language models (TEAL). ICLR 2025.
- [5] X. Men et al. ShortGPT: layers in large language models are more redundant than you expect. 2024.
- [6] Y. Li et al. EAGLE: speculative sampling requires rethinking feature uncertainty. ICML 2024.