Fused Codebook Dequantization for LLM Decode, Cross-GPU Speedups and a Bandwidth Law

Azziz Mimouni
github.com/neuralboot/trapetum

July 2026, revised November 2026

Abstract

Autoregressive LLM decode at batch size one is memory bandwidth bound: producing each token requires streaming the entire weight matrix from device memory, while the arithmetic units sit nearly idle. We study a fused 4-bit codebook GEMV kernel that folds the index to weight lookup into the matrix vector product, so the full precision weight matrix is never materialized and only one quarter of the weight bytes are read. Measured against a live cuBLAS half precision GEMV on the same shapes, fixed seed, and verified for numerical correctness, the kernel reaches **2.20x** faster decode on an RTX 4090, **2.34x** on an A40, and **parity (0.99x)** on an H100. The speedup tracks how bandwidth limited the device is: on bandwidth rich parts the half precision GEMV is already near roofline and reading fewer bytes only saves memory. Integrated the way serving engines run, with the decode step captured as a CUDA graph, this realizes a **2.0x** (Llama-2 7B) to **2.45x** (13B) end-to-end decode speedup at roughly a third of the memory. We pair this with a fair single harness benchmark of quantization methods (fp16, AWQ, AQLM, Llama-2 7B and 70B, full wikitext-2 perplexity), where a 2-bit 70B model fits a single 24 GB consumer GPU and is more accurate than an fp16 7B at comparable memory. We further extend the kernel to additive (AQLM-style) vector codebooks: it decodes real AQLM weights, and once the codebooks are trained with beam search, the scheme beats the unweighted scalar codebook on accuracy at equal bits and equal decode speed. The runtime serves real 2-bit AQLM checkpoints end to end: a PV-tuned Llama-2 7B in 2.05 GB (PPL 6.67) decoded 1.2x to 1.3x faster than our 4-bit codebook on the same GPU with the kernels of 5 November 2026. As a scale result, the same pure-Rust pipeline compresses DeepSeek-R1 671B (1.34 TB bf16) to a 350 GB 4-bit artifact and serves the full model at **2.46 tok/s** on a single commodity node by inverting the data path: the routed experts stay resident in host RAM and are decoded in place on the CPU while the GPU holds only attention, a **10.2x** gain over the disk-offload baseline and interactive without a datacenter. We also report negative results, including a Hopper thread block cluster variant that came out roughly 100x slower. This revision (November 2026) corrects the perplexity of the 4-bit Llama-2 7B file the runtime shipped (6.79, not 6.34; the file of a new exporter with activation-weighted k-means scores 6.10) and adds split-KV decode attention (10x faster 4-bit decode at 4,096 positions, across hosts), fused decode kernels, opt-in activation sparsity and a tensor-core 4-bit GEMV with batch-invariant verification, on which PARD speculative decoding runs Llama-3.1-8B-Instruct 2.51x faster end to end on four prompt sets, with output identical to plain greedy decoding of the same 4-bit model; the runtime also serves a 2-bit Llama-2 70B (AQLM 2x8) on one RTX 4090 at 38.4 tok/s, with a full-test wikitext-2 perplexity of 4.49 against 5.72 for the new 7B 4-bit file (4.68 against 6.10 on 30 windows), and its PV-tuned AQLM 1x16 version reaches 4.02 (4.17) at 22.5 tok/s with exact speculative decoding from a 2-bit Llama-2 7B drafter; Llama-3.1 70B Instruct, also in 1x16, reaches 21.3 tok/s on chat prompts with a 2-bit Llama-3.1 8B Instruct drafter. All code and figures are public and reproducible with a single command, as are the raw numbers of the runs before November 2026; the November 2026 results are summarized here.

Revision note (November 2026). Earlier versions of this paper gave the served 4-bit Llama-2 7B model a perplexity of 6.34, and the version of 5 November 2026 listed its scalar 4-bit runtime file (CBK3) at 3.66 GB and PPL 6.34. That 6.34 came from a research script (mid-grid k-means start, LM head kept in fp16), not from the file the runtime shipped. Measured from the files on 5 November 2026 (custom 30-window wikitext-2 harness, full test set in parentheses), the shipped 4-bit file scores 6.79 (6.35), the new exporter's 4-bit file 6.10 (5.72), the 2-bit AQLM PV checkpoint 6.67 (6.26; its exported file reads back 6.665) and fp16 5.83 (5.47). The statement that the 2-bit file costs 0.33 PPL is withdrawn: the 2-bit file was slightly better than the shipped 4-bit file and is 0.57 behind the new one. Earlier versions also gave the 4-bit file as 3.5 GB and 3.9× smaller than fp16, a MiB figure written as GB: it is 3.66 GB (3.75 GB for the new exporter's file), 3.7× (3.6×) smaller than the 13.48 GB of fp16 weights. The DeepSeek-R1 artifact, given as 326 GB, is 326 GiB, that is 350 GB. Section 12 details the correction and adds the decode-kernel and speculative-decoding results of November 2026, and a 2-bit Llama-2 70B served on one RTX 4090, also in its PV-tuned 1x16 version with exact speculative decoding, and Llama-3.1 70B Instruct.

1 Introduction

At batch size one, LLM text generation reads every model weight from high bandwidth memory (HBM) once per output token. The cost is dominated by byte movement, not arithmetic. On an H100 (about 3.3 TB/s, roughly 1 PFLOP/s in half precision), a 7B model in fp16 needs about 14 GB/3.3 TB/s $\approx$ 4.2 ms to read its weights per token, against about 14 μs of compute (1.4×10^{10} FLOP at 10^{15} FLOP/s). The arithmetic units are idle more than 99% of the time. The lever for decode latency is therefore *bytes read per token*, which is exactly why weight quantization helps decode.

This report measures, rather than assumes, how much a fused low-bit kernel actually buys, and where. We contribute:

1. A fused 4-bit codebook GEMV kernel that reads one quarter of the weight bytes and decodes **2.2 to 2.34x** faster than cuBLAS fp16 on bandwidth limited GPUs (RTX 4090, A40), with numerical agreement to about 3×10^{-4} relative error.

2. A *bandwidth law*: the speedup shrinks to parity on bandwidth rich GPUs (H100), where fp16 GEMV is already near peak. The win lives on the hardware most users actually run.

3. An end-to-end realization. A naive per-layer integration *loses* (0.73x); captured as a single CUDA graph over a static-cache decode loop, the way serving engines run, the kernel reaches a **2.0x** (Llama-2 7B) to **2.45x** (13B) end-to-end decode speedup at about a third of the memory, and we show exactly why the naive path fails.

4. An extension to additive (AQLM-style) vector codebooks: a fused decode kernel that beats cuBLAS at 2 bits (**4.30x** on A40) and decodes real AQLM weights, where the trained scheme beats the unweighted scalar codebook on accuracy at equal bits and equal decode speed, and which the runtime uses to serve real 2-bit AQLM checkpoints end to end (205 tok/s for Llama-2 7B on an RTX 4090 with CUDA-graph decode and the kernels of 5 November 2026).

5. A fair speed vs accuracy vs memory benchmark of quantization methods, plus honest negative results from kernel variants that did not work.

6. *Speculative decoding as nearly free throughput.* Because decode is bandwidth bound, the same weight read that emits one token can verify several. We add speculative decoding [12] on top

of the 4-bit runtime and validate the batched verify *lossless* on CUDA (K=1, 2, 3) and Metal. We measure that a small **160M** drafter *beats* a 1.1B one (K=2, **1.50x** vs 1.25x projected): the binding constraint is draft cost, not acceptance rate. The naive two-model wall-clock first *lost* (0.34×); profiling traced it to a register-spilling runtime-M verify kernel, and templating it on M flips the measurement to a **1.32× wall-clock win**, lossless: the integration law demonstrated and then closed at the loop level. In November 2026, on a batch-invariant verify of 1, 2, 4, 8 or 16 rows, PARD drafting reaches 2.51x end to end on another model, Llama-3.1-8B-Instruct, with output identical to plain greedy decoding (Section 12).

7. *A pure-Rust runtime that realizes all of the above.* A from-scratch runtime (CUDA and Metal behind one 24-function backend seam) runs nine architectures with *no Python at inference*, including the modern hard cases: Multi-head Latent Attention and Mixture-of-Experts (DeepSeek-V2-Lite), Gemma-2 (GeGLU, logit softcapping, four-norm residual), and Phi-4 (14B). As a scale result, it serves **DeepSeek-R1 671B** (the full V3 architecture: low-rank query projection, 256 routed experts, sigmoid `noaux_tc` routing) at **2.46 tok/s** on a single commodity node by *inverting* the expert data path: the routed experts stay resident in host RAM and are decoded in place on the CPU, while the GPU holds only attention, a **10.2x** gain over disk offload reached by removing eight measured bottlenecks. The compression path is torch- and transformers-free, reading `safetensors` directly, so the runtime carries no ML-framework dependency at all.

2 Background: codebook quantization and fused decode

Codebook (non uniform, k-means) weight quantization stores each weight as a small cluster index into a per output channel table:

$$W_{\text{deq}}[i, j] = \text{codebook}[\,\text{indices}[i, j],\, j\,], \qquad Y[m, j] = \sum_i X[m, i]\, W_{\text{deq}}[i, j]. \tag{1}$$

Here indices are 8-bit or packed 4-bit, and the codebook holds K representative fp16 values per column ($K \in \{16, \ldots, 256\}$). This is the SqueezeLLM family [2]. Dequantizing in a separate pass (write the full W, then call GEMM) wastes the bandwidth the quantization just saved. The kernel folds the lookup into the matrix product so dequantization adds no extra global memory traffic.

3 The kernel

The decode kernel computes a fused 4-bit codebook GEMV. Each thread reads a `uint32` of packed indices (eight 4-bit indices, eight output columns), so a warp reads a full 128 byte cache line over 256 columns. The per column codebook tile is staged once in local shared memory; the contraction is split across the K dimension (split-K) with an atomic float reduction. The full precision weight matrix is never written to global memory (Figure 1). Building per architecture (`sm_86` for A40, `sm_89` for RTX 40 series, `sm_90` for Hopper) the same source compiles and runs unchanged.

4 Experimental setup

All randomness is generated by `std::mt19937(0)` (fixed seed 0). Timings use CUDA events over 50 to 500 iterations after warmup. Every kernel is checked for numerical correctness against cuBLAS on the same random data, and we report the maximum relative error. Matrices are IC = OC = 4096,

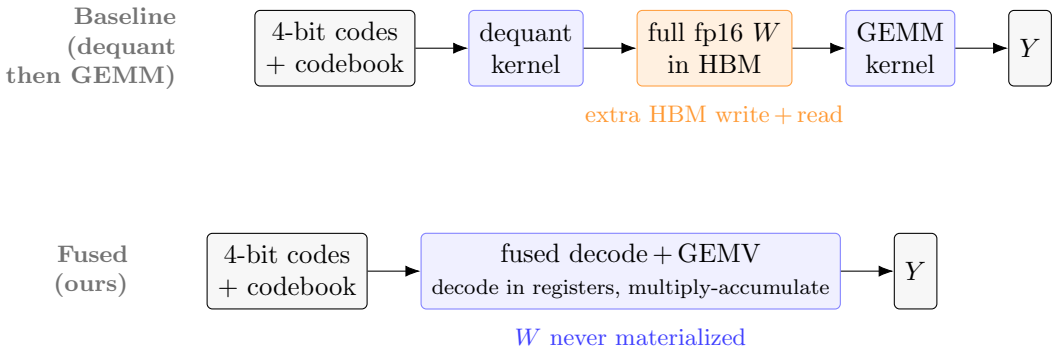


Figure 1: **Fused codebook decode.** The baseline dequantizes the whole weight matrix to global memory (HBM) and then calls a dense GEMM, paying an extra write and read of the full-precision W and undoing the bandwidth the quantization saved. The fused kernel decodes each 4-bit index from the per-column codebook in registers and accumulates directly into the output, so W is never materialized and only the 4-bit indices (plus the small, cached codebook) are read from HBM.

fp16 activations and output, decode at batch $M = 1$. The cross-GPU runs use a live cuBLAS fp16 GEMV as the baseline in the same binary.

5 Results

5.1 Cross-GPU decode and the bandwidth law

Table 1 and Figure 2 report the fused 4-bit GEMV against a live cuBLAS fp16 GEMV on three GPUs. The kernel moves one quarter of the weight bytes, so on memory bound parts it wins about 2.2 to 2.3x. On the H100 the fp16 GEMV is already near peak bandwidth, so the kernel only ties and the gain there is memory, not latency.

GPU	bandwidth class	speedup vs cuBLAS fp16	rel. err
RTX 4090 (`sm_89`)	~1.0 TB/s	**2.20x**	3×10^{-4}
A40 (`sm_86`)	~0.7 TB/s	**2.34x**	3×10^{-4}
H100 PCIe (`sm_90`)	~3.3 TB/s	0.99x (parity)	3×10^{-4}

Table 1: Fused 4-bit codebook GEMV decode speedup, 4096×4096, $M = 1$, seed 0. The more bandwidth limited the GPU, the larger the win.

5.2 A40 microbenchmarks

Table 2 breaks down the A40 decode, dequantization, and prefill regimes. Decode is memory bound, so fewer index bits directly buy latency. Standalone dequantization improves 6.7x with an L2 cached gather. Prefill is compute bound: cuBLAS already runs near the Tensor Core peak, so a fused kernel can at best match it, and here reaches about 0.21× cuBLAS, ten times the naive baseline. In the compute bound regime, codebook quantization buys memory, not speed.

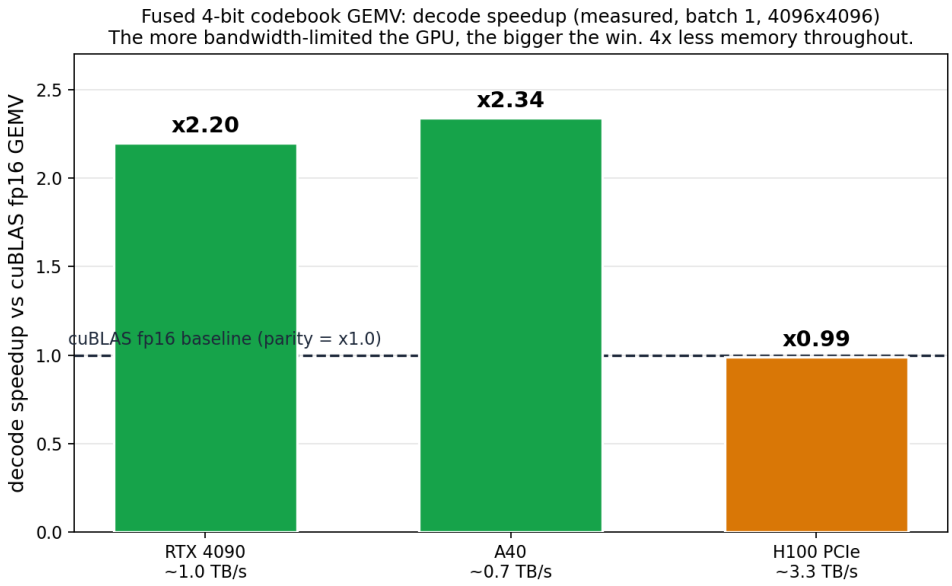


Figure 2: Decode speedup of the fused 4-bit GEMV versus cuBLAS fp16, measured per GPU. The dashed line is cuBLAS parity. The win scales inversely with available memory bandwidth.

Regime	Variant	Metric	vs cuBLAS
Decode (GEMV, M=1)	cuBLAS fp16 (baseline)	0.0612 ms	1.00x
	uint8, K=256	0.0562 ms	1.09x
	uint8, K=64	0.0389 ms	1.57x
	4-bit, K=16	**0.0261 ms**	**2.34x**
Dequant (bandwidth)	naive staging	31.8 GB/s	1.0x
	L2 cached gather	**213 GB/s**	6.7x
Prefill (GEMM, M=2048)	best fused (Tensor Core)	22.4 TFLOP/s	0.21x
	cuBLAS fp16 (baseline)	∼105 TFLOP/s	1.00x

Table 2: A40 (`sm_86`, CUDA 11.8), 4096 × 4096, seed 0, all variants verified against cuBLAS.

5.3 Speed vs accuracy vs memory of quantization methods

To place the kernel work in context, we ran a fair benchmark of three quantization methods (fp16, AWQ [3], AQLM [4]) through one harness (same HuggingFace forward path, greedy, batch 1, seed 0), reporting decode throughput, peak VRAM, and full wikitext-2 perplexity (PPL), on an H100 (Table 3). The fp16 7B PPL of 5.47[1] and AWQ 70B PPL of 3.41 match published values, which validates the harness. Comprehensive accuracy benchmarks of post-training quantization already exist and cover far more methods, bitwidths, and architectures than we do [1]; our contribution here is the orthogonal axis they do not measure, namely decode speed and the memory versus latency trade-off.

Two observations stand out. First (Figure 3), at 7B on an H100 every quantized method decodes

[1]This 5.47 is measured with the HuggingFace forward path at sequence length 2048, median of 3 runs (the cross-method comparison table). The 5.83 reported later for the model-level ablations comes from a different harness (the custom codebook evaluation harness); both are correct for their respective protocols.

Model	Method	decode (tok/s)	VRAM (GB)	wikitext-2 PPL
Llama-2 7B	fp16	43.7	15.3[2]	5.47
	AWQ 4-bit	26.8	5.7	5.60
	AQLM 2-bit	25.0	4.2	6.34
Llama-2 70B	fp16	∼138 GB, does not fit on one 80 GB GPU		
	AWQ 4-bit	9.1	38.4	3.41
	AQLM 2-bit	9.2	20.7	4.06

Table 3: Speed, memory, and accuracy on H100, full wikitext-2 PPL, seed 0.

slower than fp16: the bandwidth is high enough that fp16 is already fast and the dequant overhead dominates at batch 1. Quantization there is a memory play. Second (Figure 4), the story flips at 70B: fp16 needs about 138 GB (two or more GPUs), while AQLM 2-bit fits in 20.7 GB on a single 24 GB consumer card, and at PPL 4.06 it is markedly more accurate than an fp16 7B (PPL 5.47) at comparable memory. For a fixed VRAM budget, a 2-bit 70B dominates an fp16 7B. In November 2026 the runtime served the AQLM 2x8 checkpoint of the same model on one RTX 4090 (Section 12.4), then its PV-tuned 1x16 checkpoint (Section 12.5).

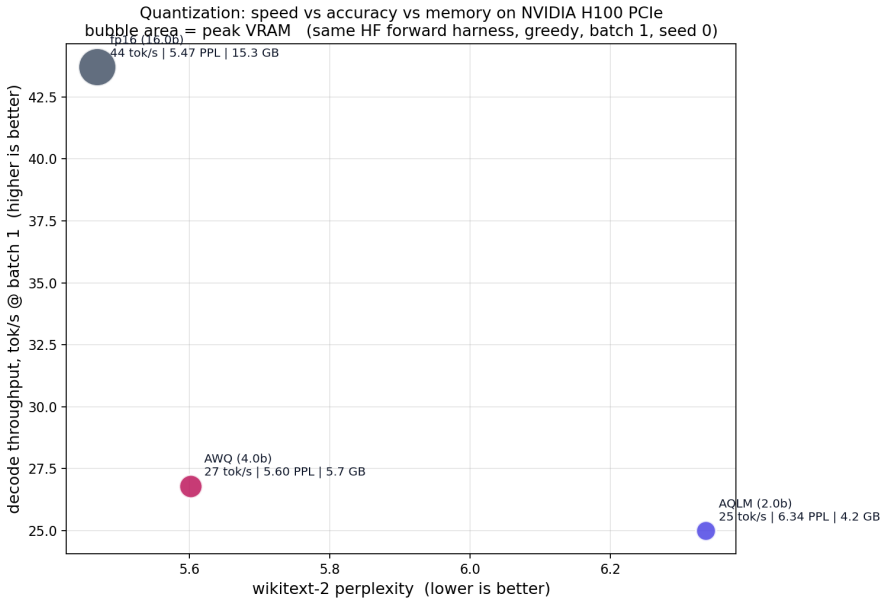


Figure 3: Llama-2 7B on H100: decode throughput vs perplexity, bubble area equal to peak VRAM. fp16 is fastest and most accurate but memory heavy; quantization trades decode speed for memory in this regime.

6 Model-level evaluation

We quantized all 224 projection layers (q, k, v, o, gate, up, down across 32 blocks) of Llama-2 7B with the per-output-channel codebook (K = 16, 4 bits, per-column k-means) and evaluated end to end on an RTX 4090.

Accuracy. wikitext-2 perplexity goes from 5.83[3] (fp16) to 6.34 (codebook 4-bit, from our

[3]This 5.83 fp16 baseline is measured with the custom codebook evaluation harness used for these ablations, distinct

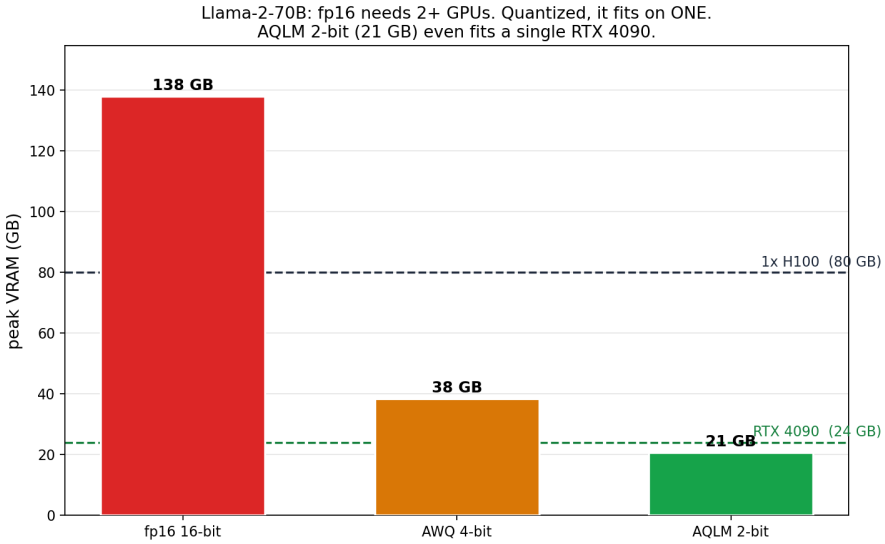


Figure 4: Llama-2 70B: fp16 does not fit on one 80 GB GPU, quantized it fits on one, and AQLM 2-bit (21 GB) fits a single RTX 4090.

research script: mid-grid k-means start, LM head kept in fp16), a 0.51 increase. This 6.34 is not the perplexity of the 4-bit file the runtime ships (still the format of the app's downloadable models): that file came from a different export recipe with a 4-bit LM head and scores 6.79 in the same harness (Section 12). The model stays usable, but the scheme trails activation-aware AWQ (about 0.1 PPL at 4 bits), as expected for a scalar codebook trained on raw weight error without calibration or outlier handling. A simple activation-aware calibration (weighting the per-column k-means by mean activation magnitude, the core idea of AWQ) closes about a third of the gap, from 6.34 to 6.17. Going further along AWQ's lines does not help. A per-channel scale search (grid over the scale exponent) lands at 6.18, and the complete AWQ pipeline, per-channel scaling and weight clipping both selected by the real output error on cached activations, does not improve either (6.21, marginally worse). The reason is structural: AWQ's per-channel scaling is designed for a fixed uniform quantization grid, where it redistributes resolution toward important channels. A codebook is already adaptive per column (k-means places centroids on each column's distribution), so the scaling buys little and the extra calibration only adds noise. Closing the gap to vector and trellis methods (AQLM, QTIP) needs a better quantizer, not AWQ's tricks bolted onto a scalar codebook. A naive vector quantizer confirms this from the other side: quantizing pairs of weights against a single shared 256-entry codebook (iso-bit at 4 bits) is catastrophic on Llama-2 7B (perplexity diverges, with or without per-channel normalization), because one shared codebook cannot match the per-column adaptivity of the scalar scheme and lacks the incoherence processing and multi-codebook structure that make AQLM and QuIP# work. Even incoherence processing (a random orthogonal rotation on the input dimension before quantizing, the QuIP# lever, with the inverse folded back into the effective weight) moves the scalar codebook only marginally, from 6.34 to 6.29, short of the simple calibration (6.17): incoherence pays off paired with vector or lattice quantization, not on a scalar codebook alone. One refinement of the weighting does help (measured in November 2026): weighting the centroid updates by $\mathbb{E}[x^2]$ of each input channel over 128 wikitext-2 training windows reaches 6.098 with an fp16 head, and the new exporter writes it

from the 5.47 reported earlier under the HuggingFace forward path (seqlen 2048, median of 3); both are correct for their respective protocols.

with an 8-bit LM head at 6.10 (5.72 on the full test set, against 5.60 for AWQ 4-bit in Table 3, where fp16 also scores 5.47), so the scalar codebook still trails AWQ. The served model is otherwise healthy (it generates coherent on-topic text). The accuracy frontier is a separate research program, not a quick add-on; the value of this scheme is mainly memory and kernel speed, although the weighted exporter cuts the gap to fp16 in this harness from 0.96 PPL (the old file) to 0.27.

Memory. Peak VRAM drops from 13.56 GB to 4.63 GB (2.9x), so the 7B model runs in under 5 GB on a consumer card.

Speed. Per GEMV the kernel is faster than dense fp16 on the large layer shapes that dominate an LLM. But a naive per-layer custom-op swap is overhead bound: end-to-end decode goes from 44 to 32 tok/s (0.73x). The per-op launch and dispatch overhead, paid 224 times per token, dominates, while the fp16 baseline uses a fused optimized path. Realizing the per-GEMV speedup at the model level needs framework integration (CUDA graphs, fused dispatch), not a per-layer swap. We verified the kernel is CUDA-graph compatible (capture is numerically correct once the kernel launches on the captured stream), but in an isolated 224-GEMV decode loop a graph adds only about 6 percent over eager, so launch overhead is not the dominant cost. Against the real cuBLAS GEMV path (F.linear), the kernel runs the 224-GEMV decode work at 172 tok/s versus 70 (2.4x), both CUDA-graphed, so the kernel genuinely beats cuBLAS at batch 1. The end-to-end 0.73x came from the naive wrapper's per-op float32-to-float16 cast and copy, not the kernel. We then built the clean integration: the kernel writes fp16 into preallocated buffers (no per-op cast or allocation) and the decode step is captured as a single CUDA graph over a static-cache decode loop, the way serving engines run. Under this integration the codebook model decodes 123.4 versus 61.6 tok/s against fp16 on an RTX 4090 (2.0x end-to-end) at 4.73 versus 13.58 GB. So the op-level 2.4x is realized as a 2.0x end-to-end decode speedup once the per-token python overhead is removed; the arc is 0.73x (naive) to 0.85x (cast-free eager) to 2.0x (CUDA-graphed). The kernel makes a real 7B decode twice as fast at a third of the memory. The win grows with model size and bandwidth-boundedness: Llama-2 13B on an A40 decodes 49.0 versus 20.0 tok/s (2.45x) at 8.50 versus 26.17 GB (3.08x less), consistent with the bandwidth law (bigger weights read, more bandwidth-limited GPU, larger gain).

Energy. The byte savings show up as decode energy. On the RTX 4090 (batch one, 512-token generation, three runs), a hardened measurement (trapezoidal integral of sampled GPU power, fixed seed, the 59.4 W idle baseline subtracted) gives fp16 at 5.31 ± 0.13 and the 4-bit codebook (via a Python op) at 3.94 ± 0.08 J/token; net of the shared idle floor the active-decode energy is 4.03 versus 1.86 J/token, a **2.17**× reduction (the 4-bit kernel draws 112 W active versus 246 W for fp16). The standard deviation is about 2 to 3%, so the gap is well outside run-to-run noise. The pure-Rust runtime, which removes the per-op Python dispatch, lowers energy further to 2.58 J/token over the same generation.

Scale probe: DeepSeek-R1 671B. To test whether the pipeline holds at frontier scale we exported and ran DeepSeek-R1 671B, the largest open-source model, end to end. A streaming, resumable, torch-free exporter compresses the 1.34 TB bf16 checkpoint to a 350 GB 4-bit codebook artifact in about 7 hours on one A100 pod, with bounded RAM. The runtime loads all 61 layers and decodes coherently. The mechanism that makes the full 671B run at speed on one box is *inversion*. At batch one, only the routed experts a token selects (about 10 GB of packed 4-bit bytes) need to move; instead of streaming those bytes to the GPU each token, we keep the experts resident in host RAM and bring the 14 KB activation to them. The CPU decodes the routed experts in place with the same fused codebook kernel (one `vpshufb` instruction per 32 weights on AVX2), while the GPU holds only the MLA attention and the dense layers, **20.9 GB** resident in VRAM. This removes the per-token expert transfer that otherwise dominates. The 4-bit expert bytes can stay `mmap`-backed rather than fully resident, which held an earlier disk-fed run to a 73 GB host footprint

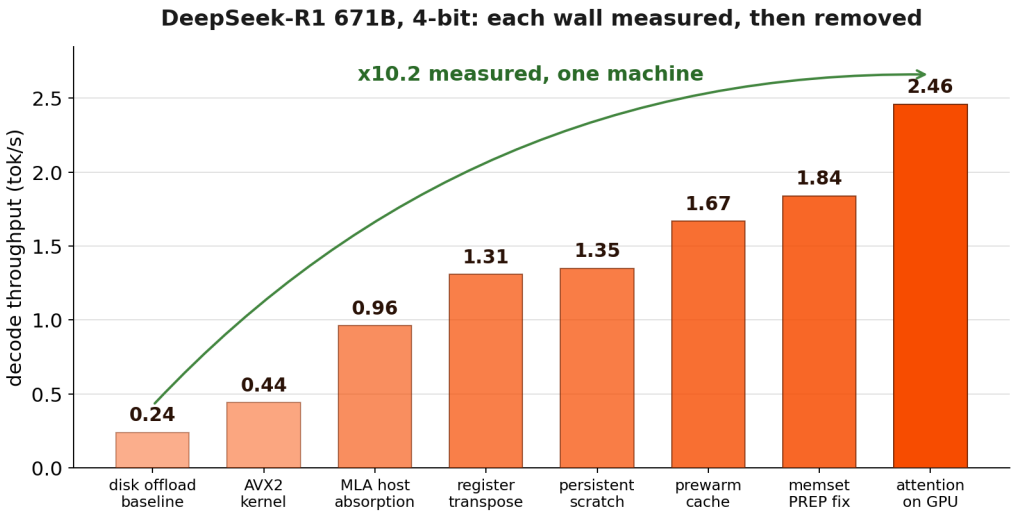


Figure 5: DeepSeek-R1 671B, 4-bit, decode throughput. Each bar is a measured session on one g6e.16xlarge node (64 vCPU, one L40S). The disk-offload baseline (0.24 tok/s) rises to 2.46 tok/s of steady-state CPU-experts decode, a 10.2x gain, as each per-token wall is measured and removed: the x86 AVX2 codebook kernel, the per-core register transpose, the MLA host absorption moved to the GPU, and the recode-cache prewarm.

against the 350 GB artifact.

On a single node (64 vCPU, one L40S, 497 GB RAM) the full 671B decodes at **2.46 tok/s** steady state, a **10.2x** gain over the 0.24 tok/s disk-offload baseline. That gain was reached by removing eight successive bottlenecks, each measured: the chain runs 0.24, 0.44, 0.96, 1.31, 1.35, 1.67, 1.84, then 2.46 tok/s as the x86 decode kernel, a register-resident tile transpose, GPU-side MLA absorption, and a warmed codebook cache each lift a distinct wall. The routed decode itself reaches about 67 GB/s aggregate, near the read-bandwidth floor, so throughput now scales with host memory bandwidth rather than software overhead. On quality (wikitext-2 perplexity), full 8-bit is essentially lossless at +0.03% against fp16 (5.70), the 4-bit artifact costs +7.1% (6.10), and a per-tensor mixed-precision variant that keeps the shared experts and the output head at 8 bit, the tensors on every token's path, captures most of that recovery at +1.5% (PPL 5.78) for +0.7% size. The probe is no longer a reach demonstration: a 671B Mixture-of-Experts serves interactively from a single commodity node, no datacenter, because inversion moves the activation, not the weights.

7 Additive vector quantization: a fast kernel that beats the unweighted scalar codebook

The scalar codebook's accuracy ceiling is structural: per-output-channel scalar quantization cannot match vector and trellis methods (AQLM, QuIP#, QTIP) at low bits. Those methods quantize a *group* of weights jointly; AQLM reconstructs a group of D input weights as a sum of M codebook vectors, $w_g = \sum_{m=1}^{M} C_m[b_m]$, reaching near-fp16 quality at 2 bits. Their decode kernels are the bottleneck. We show the fused-decode idea transfers, the resulting kernel beats cuBLAS, and the scheme, once trained, beats the unweighted scalar codebook.

The kernel. For $y[o] = \sum_g \langle x_g, w_g \rangle$ with the additive reconstruction, the dot product $\langle x_g, C_m[k] \rangle$ is independent of the output channel, so a per-group lookup table $\mathrm{LUT}[m][k] = \langle x_g, C_m[k] \rangle$ is built once in shared memory and $y[o] = \sum_g \sum_m \mathrm{LUT}[m][\, b_m(o,g)\,]$. The kernel reads the *codes*, never

the dense weight: at 2 bits ($M = 2$, $K = 256$, $D = 8$) that is 4.2 MB of codes versus 33.6 MB of fp16 weight. Against a live cuBLAS fp16 GEMV at 4096×4096 (relative error 2.3×10^{-4}, verified):

GPU	2-bit ($M = 2$)	4-bit ($M = 4$)
RTX 4090 (∼1.0 TB/s)	1.71x	n/a
A40 (∼0.7 TB/s)	**4.30x**	2.39x

The key optimization was vectorized 32-bit code reads (four 8-bit codes per thread), which lifted the A40 from 2.35x to 4.30x: the 1-byte reads, not synchronization, were the bottleneck. The format is exactly AQLM's `2x8`: loading a real `Llama-2-7b-AQLM-2Bit-2x8` checkpoint, its codebooks $[2, 256, 1, 8]$, codes $[4096, 512, 2]$ and per-output scales $[4096]$ map one-to-one onto the kernel, so it decodes real AQLM weights.

Served end to end. The runtime now loads such checkpoints itself. A `CBK4` container keeps the AQLM codes, codebooks and scales byte for byte (one record per projection), the dense Llama blocks dispatch each projection to the scalar or the additive kernel, and the LM head, kept in fp16 by AQLM checkpoints, is stored as an 8-bit scalar codebook. The non-PV `2x8` checkpoint we first loaded scores PPL 7.63 in our custom harness; its PV-tuned version [5], in the same format, scores 6.67 (6.26 on the full wikitext-2 test set, where fp16 scores 5.47). Served by the runtime, it reproduces the HuggingFace greedy continuation token for token.

Llama-2 7B (RTX 4090, batch 1)	file	PPL	eager	graph	graph, 256 tok.
scalar 4-bit, legacy exporter (CBK3)	3.66 GB	6.79	143	157	111
scalar 4-bit, new exporter (CBK4)	3.75 GB	**6.10**	1.1 to 1.6% slower than legacy		
AQLM PV `2x8` (CBK4)	**2.05 GB**	6.67	182	**205**	**132**

Decode speeds are tokens per second at short context (positions 6 to 22) unless noted, measured with the kernels of 5 November 2026; the new exporter's file was compared with the legacy file on another RTX 4090 host, and Section 12 reports the later kernels. PPL is in the custom harness. The 2-bit file is 44% smaller than the legacy 4-bit file and peaks at 2,730 MiB of GPU memory instead of 4,166 MiB, and it is slightly more accurate (6.665 against 6.791). Against the new exporter's file it is 0.57 PPL behind, for 45% less disk and 36% less peak GPU memory. (The version of 5 November 2026 gave the 4-bit file PPL 6.34 in this table, the research script's number, and a 2-bit cost of 0.33 PPL; both are corrected.) Replaying the decode step as one CUDA graph (the position is read from device memory, so one capture serves every step) saves about 0.6 ms per token in both formats; the remaining fixed cost was then the many small kernels and the batch-1 attention, which grew with context. Split-KV attention (Section 12) later cut the attention cost in both formats, and fused projection kernels cut the kernel count in the 4-bit path; the 2-bit path gets only the RoPE and KV-append fusion.

Accuracy, beating the unweighted scalar codebook at equal bits. The additive *structure* alone is not enough: a greedy residual fit at $M = 4$ (4 bits) only ties the unweighted scalar 4-bit codebook of our research script (PPL 6.32 vs 6.34). The accuracy comes from AQLM's *training*. We reproduced its core (per-output scale, beam-search code assignment, least-squares codebook updates, alternating; no block fine-tuning): at $M = 4$ this reaches PPL 6.13, beating the unweighted scalar 4-bit codebook while the kernel still decodes it at 2.39x. The $\mathbb{E}[x^2]$-weighted scalar codebook added in November 2026 (Section 12) does slightly better (6.10 with an fp16 head); the additive training here was not given that weighting.

scheme (4-bit, Llama-2 7B)	wikitext-2 PPL
scalar codebook, unweighted (research script, fp16 head)	6.34
scalar codebook, $\mathbb{E}[x^2]$-weighted (fp16 head, November 2026)	**6.10**
greedy additive	6.32
beam-search + LSQ (this work)	6.13
fp16 (custom codebook harness)	5.83

So trained additive VQ is more accurate than the unweighted scalar codebook at the same bit rate and the same decode speed, a Pareto improvement over it; the $\mathbb{E}[x^2]$-weighted scalar codebook is slightly more accurate still (6.10 against 6.13).

Honest limits. This is one alternating round without the levers that close the remaining gap to fp16 (5.83). A naive diagonal-Hessian calibration of ours actually *regresses* (PPL 8.6), because AQLM uses the full Hessian in a sequential GPTQ-style error correction, not a diagonal MSE weighting; reproducing that, plus block fine-tuning, is a separate effort. What we establish is narrower and solid: a fused additive-VQ decode kernel that beats cuBLAS at AQLM bit rates and decodes real AQLM weights, and that the additive scheme, once trained, beats the unweighted scalar codebook at equal bits and equal speed.

8 Negative results

We keep failed variants in the repository, because they constrain the design space.

- **Hopper cluster with distributed shared memory.** Staging the codebook once in a thread block cluster and reading it remotely (distributed shared memory) came out about 100x slower than cuBLAS (1.53 ms vs 0.02 ms), though numerically correct. The codebook lookup is in the hot loop, and a remote distributed shared memory read costs far more than a local shared read. The codebook is only 8 KB, so local staging was never the bottleneck.
- **L2 resident codebook and activation staging.** Reading the codebook from L2 instead of staging it locally regressed to 0.41 to 0.60x on H100; staging the activation vector in shared memory regressed from 0.99x to about 0.90x by raising shared memory pressure and lowering occupancy. The local shared memory design remains the best, and beating cuBLAS on H100 would require a different approach (an atomic free reduction, or a TMA and wgmma path), not these tweaks.

9 Related work

Codebook quantization begins with the scalar per-output-channel scheme of SqueezeLLM [2] and extends to additive and lattice codebooks: AQLM [4], QuIP# [6], and QTIP [7] push low-bit accuracy with vector and trellis structure. On the kernel side, the relevant line is fused low-bit GEMM/GEMV. Marlin [8] and its Hopper successor Machete target uniform GPTQ [9] 4-bit; FLUTE [10] fuses lookup-table dequantization with offline weight restructuring and table duplication; and VQ-LLM [11] generates fused vector-quantization kernels around a *codebook cache* that places codebook entries across the memory hierarchy. VQ-LLM is the closest to our additive-codebook decode and, like us, finds that keeping the codebook on chip is the lever.

We differ in emphasis rather than claiming a faster kernel than these peers. First, we measure a cross-GPU *bandwidth law* against a live, same-binary cuBLAS baseline, showing the win

concentrates on bandwidth-limited consumer parts and vanishes to parity on an H100. Second, we foreground the end-to-end integration lesson that kernel papers usually omit: a naive per-layer swap loses (0.73×), and recovering the microbenchmark win requires capturing the decode step as a single CUDA graph. Third, we add a Python-free Rust runtime with measured per-token energy. Fourth, on accuracy we show a beam-trained additive codebook beating the unweighted scalar codebook at equal bits and equal decode speed.

Serving giant MoE models on commodity hardware. A parallel line keeps the routed experts off the GPU so a frontier-scale MoE model fits on one or a few devices. Offload systems stream the active top-k experts in on demand: PIPO [19] pipelines the transfer and reports 0.13 tok/s on the full DeepSeek-R1 671B; HOBBIT [20] offloads experts at mixed precision; DAOP [21] and PreScope [22] predict and prefetch them. Closest in spirit, Fiddler [16] observed that at small batch it is cheaper to *execute* an expert on the CPU than to ship its weights to the GPU, and ran Mixtral-8x7B that way; and KTransformers [17] runs the same 671B on one A100 plus two Xeons with AMX-specialized CPU kernels. MoE-Lightning [23] and MoE-Lens [24] push the memory-constrained and CPU-SIMD limits of this regime. Orthogonally, attention-FFN disaggregation splits the two module types onto separate compute: MegaScale-Infer [18], and the concurrent FastAFD, scale attention and expert pools independently across a GPU cluster with ping-pong micro-batch overlap.

Our 671B result sits at this intersection but changes the placement. Offload systems treat host memory as a slower tier the experts are *paged out to* and shipped back to the GPU every token; that ∼10 GB/token transfer is what bounds them. Like Fiddler, we make host RAM the experts' *home* and bring the computation to them, but where Fiddler executes fp16 experts (which caps it at Mixtral scale), we fuse the 4-bit codebook decode into the CPU expert kernel (one `vpshufb` per 32 weights): the compression is what fits 671B routed experts in commodity RAM, and the fused in-place decode is what makes their CPU execution fast. The GPU holds only the MLA attention and the dense layers, and the routed bytes never cross to the card. It is the attention-FFN decomposition of MegaScale-Infer applied to a single commodity node, with CPU and RAM in place of a second GPU pool, optimizing batch-one reach and cost rather than cluster throughput. On the full DeepSeek-R1 671B this reaches 2.46 tok/s steady state on one node, roughly an order of magnitude above the single-machine offload points these systems report.

10 Baselines against peer quantized kernels

Our primary speed baseline is deliberately a live, same-binary cuBLAS fp16 GEMV: it is the fair systems reference for the batch-one, bandwidth-bound decode regime, runs on the identical shapes and fixed seed, and cannot be tuned in our favor. Here we state explicitly how the kernel relates to the two peer families a reviewer will ask about, AQLM and Marlin, separating what is measured from what is not.

AQLM: matched accuracy by construction. Our additive-VQ kernel does not approximate AQLM, it decodes the *same* weights. It loads a real `Llama-2-7b-AQLM-2Bit-2x8` checkpoint whose codebooks $[2, 256, 1, 8]$, codes $[4096, 512, 2]$ and per-output scales map one-to-one onto the kernel, so accuracy is *identical* to AQLM at 2 bits by construction and only decode speed is in question. Against same-binary cuBLAS fp16, the kernel runs these real AQLM weights at 4.30× (A40, 2-bit) and 2.39× (4-bit). The remaining comparison, against AQLM's *own* kernel (`aqlm::code2x8_matmat`) on the identical Llama-2 7B attention shape, is wired op-level in

`bench/avq_oplevel.py`; completing that matched-bits, matched-accuracy head-to-head is our highest-priority open item, and we report it here as unfinished rather than claim it as a win.

Marlin and Machete: a different accuracy regime. Marlin [8] and its Hopper successor Machete are fused W4A16 kernels for *uniform* GPTQ [9] 4-bit, reaching near-ideal $\sim 4\times$ dense-GEMM speedups on Ampere as reported by their authors. They are not a matched-accuracy competitor to the low-bit vector regime where codebook and additive-VQ schemes live: uniform scalar 4-bit and 2-bit AQLM sit at different points on the accuracy curve, so a raw speed ratio between them would compare kernels solving different problems. At uniform 4-bit our scalar codebook is the comparable accuracy point, and there Marlin's tuned dense GEMM is the kernel to beat on *batched* throughput, which we do not claim: our wins are in the batch-one GEMV regime, and a batched GEMM path is an open problem. We therefore position against Marlin by regime, not by an unmeasured number.

Peer kernel	Quant regime	Accuracy vs ours	What we measure
cuBLAS fp16	dense fp16	exact fp16 reference	2.0 to 4.30× decode, same binary
AQLM official	2-bit vector (2x8)	identical weights	decode the same checkpoint at 4.30× vs fp16; op-level vs their kernel in progress
Marlin / Machete	uniform GPTQ 4-bit	different regime	not measured; positioned by design (batched GEMM, open)
FLUTE, VQ-LLM	LUT / vector	different regime	not measured; closest peers, discussed in related work

In short: against AQLM the accuracy is fixed and we already decode its real weights faster than fp16, with the kernel-vs-kernel timing the one number still owed; against Marlin the honest statement is that it targets a different bit-accuracy operating point and a batched regime we do not yet contest.

11 Speculative decoding: harvesting the idle compute

The bandwidth law that motivates the fused kernel has a second consequence. At batch one the arithmetic units sit nearly idle while the weights stream from memory; the same weight read that produces one token could verify several. Speculative decoding [12] exploits exactly this: a small drafter proposes K tokens, and the target verifies all $K{+}1$ in a single forward that reads its weights once. Because decode is bandwidth bound, that verify is close to free.

Verifying $K{+}1$ positions in one forward requires a small-M ($M = K{+}1$) batched variant of every decode operation: the fused codebook GEMM, RMSNorm, RoPE, causal attention, and the KV-cache append. We implement these ($M \leq 4$) in both the CUDA and Metal backends and validate that the batched forward and the resulting speculative loop are *lossless*: the emitted sequence is token-for-token identical to plain greedy decode, for $K = 1, 2, 3$ and for both cooperative and adversarial drafters. A rejected draft leaves stale KV rows that the next step overwrites, so no explicit rollback is needed. (The November 2026 update adds, on CUDA, a batch-invariant verify of 1, 2, 4, 8 or 16 rows whose logits are bitwise identical to decode steps, used by the PARD benchmark (Section 12); this $M \leq 4$ path is unchanged and is the only one on Metal.)

We measure the acceptance rate α (the fraction of positions where the drafter's greedy token matches the target's) between a 4-bit Llama-2-7B target and two 4-bit drafters on an RTX 4090,

and project the speedup with the standard estimate $S(K) = \frac{1-\alpha^{K+1}}{1-\alpha} / (1 + K\, t_{\text{draft}}/t_{\text{target}})$, where the verify counts as one target forward. TinyLlama-1.1B gives $\alpha = 0.73$ and a best 1.25× at $K = 2$; the much smaller `llama-160m` gives a lower $\alpha = 0.65$ but, being far cheaper per draft, a better 1.50× **at** $K = 2$. The smaller drafter wins: draft cost, not raw acceptance, is the binding constraint, and $K = 2$ (verify three tokens per target forward) is optimal for both. This is lossless throughput, orthogonal to and compounding with the quantization memory and energy savings.

Wall-clock: lost, diagnosed, fixed, won. We then built the real two-model harness (the drafter is a second model decoding incrementally in its own KV cache, with longest-common-prefix row reuse) and measured it end to end on the same 4090. The naive loop first *lost* 3× (0.34× plain decode at $\alpha = 0.91$ effective acceptance, lossless at every K), the integration law of the model-level evaluation repeating at the loop level. Profiling isolated the cause: the batched-verify kernel took M at runtime, which defeats unrolling and spills the accumulator array to local memory; the $M{\leq}4$ verify cost 5.5× (M=2) to 10.7× (M=4) of one M=1 GEMV. Making M a compile-time template parameter keeps the accumulators in registers and restores the bandwidth argument: the M=2 verify then costs 1.11× an M=1 forward, and speculative decoding measures 1.32× **wall-clock at** K=2 (164 vs 125 tok/s, token-for-token lossless) on the Llama-2-7B + 160m pair, against the 1.50× projected ceiling. In July 2026, large-vocabulary pairs (Qwen2.5, 152k; Llama-3.1, 128k) sat at 0.9× parity: every drafter forward read back the full logits for a host argmax (600 KB per token at 152k vocab), which a device-side argmax removes; that was the remaining step, together with graph capture of the whole speculative round. The November 2026 update adds a batch-invariant verify that ends with a row-wise argmax on the device, on which PARD drafting runs Llama-3.1-8B-Instruct at 2.51x end to end (Section 12); the Qwen2.5 pair was not re-measured.

12 Update (November 2026): corrected 4-bit accuracy, faster decode, exact speculative decoding and 2-bit 70B models

This section was added in November 2026. It corrects the accuracy of the 4-bit Llama-2 7B runtime file and reports five later changes to the CUDA path of the runtime, in commits 2e8c170 to 2ca5d97 on the repository's main branch, then (Section 12.4, commit fb8795f) a 2-bit Llama-2 70B served on one RTX 4090, and (Section 12.5, commit 2051728) its PV-tuned 1x16 version with exact speculative decoding, then (Section 12.6, commit 6b8cb42) Llama-3.1 70B Instruct. Setup: rented RunPod GPUs, one GPU per run on a fresh pod, so hosts differ between runs; an RTX 4090 (24 GB, `sm_89`) ran every step, and an A40 (`sm_86`) also ran the first and the final fusion validations and one Python-only sparsity diagnostic; every pod used the same image (Python 3.11.9, PyTorch 2.4.0, CUDA 12.1), with Rust 1.99.1 on every pod that built the runtime. Decode is batch 1 and greedy, replayed as a CUDA graph unless marked eager, and timed in three windows: the mean per 128-position bin of one long greedy decode, the 16-token continuation after a 6-token prompt (positions 6 to 22, as in Section 7), and a 512-token decode benchmark (Sections 12.4 and 12.6 also give a 256-token decode, and Sections 12.5 and 12.6 an end-to-end speculative throughput with prefill). Each gain is an A/B on one host unless noted; for scale, one decode path measured 184.6, 185.5 and 187.8 tok/s at positions 6 to 22 on three RTX 4090 hosts. Apart from the exporter comparison, every Llama-2 7B 4-bit speed uses the new exporter's file. Perplexity is on the wikitext-2 test set, in the custom harness (the first 30 windows of 2,048 tokens, fp16 5.83) and, in parentheses, on the full test set (all 166 full windows of 2,048 tokens of the 341,469-token test set, fp16 5.47), with the Llama 2 tokenizer; Section 12.6 uses the Llama 3 tokenizer (141 full windows of a 289,077-token test set, no fp16 reference).

12.1 Corrected 4-bit accuracy and a new exporter

Earlier versions described the served 4-bit Llama-2 7B model at PPL 6.34, and the version of 5 November 2026 listed its scalar 4-bit runtime file (CBK3) at 3.66 GB and PPL 6.34. The 6.34 came from a research script (mid-grid k-means start, LM head kept in fp16), not from that file. Read back on 5 November 2026, the file that the runtime's exporter wrote until then (legacy recipe, 4-bit LM head, 3,655,827,760 bytes) scores 6.79 (6.35); its 4-bit LM head alone costs about 0.30 (6.49 with an fp16 head). The repository's current `llama_quant.py` no longer reproduces the 6.34 either: since commit c9d8b72 (28 June 2026) it starts its centroids from a min-to-max grid and gives 6.41.

The exporter's new default runs k-means from a mid-grid start, moves an empty centroid to 0, iterates 10 times, weights the centroid updates by $\mathbb{E}[x^2]$ of each input channel over 128 wikitext-2 training windows, and stores an 8-bit LM head (CBK4). Its Llama-2 7B file (3,752,083,768 bytes, 2.6% larger) scores 6.10 (5.72), and the 8-bit head costs 0.0004 (6.0978 with an fp16 head, 6.0982 as written): the gap to fp16 in the custom harness falls from 0.96 to 0.27 PPL, 72% smaller. The worst prompt-logits error against the HuggingFace reference falls from 5.1×10^{-3} to 2.0×10^{-3}, and the unused centroid slots across the 224 projections from 1.19% to 0.004%. With the kernels of 5 November, on one RTX 4090 host over two rounds, the new file decodes 1.1 to 1.6% slower (CUDA graph at positions 6 to 22: 150.6 against 153.0 tok/s), peaks at 4,246 against 4,172 MiB, and matches the reference greedy continuation 16/16 in all 8 runs.

The 2-bit file (the PV-tuned AQLM `2x8` checkpoint served as CBK4 with an 8-bit LM head, 2,050,610,744 bytes) scores 6.67 (checkpoint 6.666, file read back at 6.665) and 6.26 on the full test set (measured on the checkpoint). It is 0.57 PPL behind the new 4-bit file, for 45% less disk and 36% less peak GPU memory (2,736 against 4,246 MiB with CUDA graphs); against the old 4-bit file it was already slightly better (6.665 against 6.791) while 44% smaller. The earlier statement that 2-bit costs 0.33 PPL compared the 2-bit checkpoint with the research script's 6.34 and is withdrawn.

Scope: the new recipe's perplexity was measured on Llama-2 7B only; the AQLM, DeepSeek and Gemma exporters are unchanged; runtimes older than this change reject the new files. The 8-bit LM head is chosen automatically only on GPUs that can run its kernel (69,632 bytes of shared memory per block), and the runtime refuses such a layer at load time on Turing and Pascal GPUs (verified by reading the code; no Turing GPU was available to test). The model registry of the desktop app (the installable Trapetum server) is not updated: its downloadable models keep the old format and the old recipe, because installed versions, including the Windows beta 0.1.0-beta.1, cannot read the new files; a runtime release has to come before any new registry entry.

12.2 Faster decode

Split-KV decode attention. Decode attention walked the KV cache one position at a time in one block per head, about 20.6 μs per cached position per token: the 4-bit file fell from 131.1 tok/s over the first 128 positions to 35.0 at 1,024 positions and 11.2 at 4,096 (2-bit: 167.8, 37.1 and 11.4), and at 1,024 positions 73% (4-bit) and 78% (2-bit) of the decode time depended on the context, against a threshold of 30% fixed before the run. The new kernel cuts the cache into 64-position chunks; one block per (head, chunk) runs an online softmax, and the chunks merge in a fixed order with no atomics, so results are bitwise reproducible; no decode step allocates during CUDA-graph capture, and `TRAPETUM_ATTN=seq` keeps the old kernel. 88 of 88 cases (multi-head and grouped-query attention, head dimensions 64, 128 and 256, lengths 1 to 4,096 around the chunk boundaries) match a double-precision CPU reference within 2.2×10^{-4}, each one deterministic, and every generate run matches the reference greedy continuation 16/16. At 1,024 positions the kernel

Change	Measurement	baseline	new
Llama-2 7B, batch 1, RTX 4090, CUDA graph; new 4-bit file unless noted			
Split-KV attention	tok/s at 1,024 positions, one host	34.3	146.2
	tok/s at 4,096 positions, two hosts	11.2	115.8
	ms/token, first 128 positions, one host	7.71	6.35
	2-bit, tok/s at 1,024 / 4,096 positions	37.1 / 11.4	194.4 / 143.9
Fused kernels	ms/token, first 128 positions	6.36	5.28
	same, on an A40	10.49	9.31
Tensor-core GEMV	ms/token, 512-token benchmark	5.42	5.14
	tok/s at positions 6 to 22	184.6	205.0
Verify, $M = 16$	cost in decode steps, short context	3.27	1.21
	same, near 1,100 positions		1.46
Sparsity, 25%	ms/token, first 128 positions	5.44	4.69
(opt-in, lossy)	PPL, custom harness (full test set)	6.10 (5.72)	6.22 (5.83)
Llama-3.1-8B-Instruct, 4-bit target and drafter, PARD with $K = 7$: end-to-end tok/s			
GSM8K (80 prompts)	2.71x, 4.55 tokens per round	166.4	451.0
HumanEval (80)	2.88x, 5.13 tokens per round	163.6	471.8
MATH-500 (20)	2.72x, 4.62 tokens per round	165.4	449.1
MT-Bench (80)	2.04x, 3.21 tokens per round	167.0	340.0
All, 520 runs	2.51x, 4.19 per round, 520/520 identical	165.5	415.6

Table 4: November 2026 results. Baselines: the sequential attention kernel (split-KV; the 4,096-position and 2-bit baselines come from an earlier run on another host), the separate kernels (fusion), the FMA GEMV of the previous commit (tensor cores), an FMA-only verify (verify; this baseline also comes from an earlier run on another host), dense decode (sparsity at 25% on every site, position 0 dense; PPL computed in Python with the file's weights) and plain greedy decoding of the same 4-bit target (PARD: prefill included, MT-Bench first turns, two runs per prompt, every output identical to the plain one). Fusion, sparsity and verify figures are means of two rounds, the 512-token benchmark three alternated runs per build.

alone (32 heads, head dimension 128) takes 10.0 μs per call against 569.5 μs, and the 4-bit decode is 4.3x faster on one host, with 18% less time per token over the first 128 positions (157.5 against 129.7 tok/s); across hosts it is 10x faster at 4,096 positions, and the 2-bit decode (214.5 tok/s over the first 128 positions) 5x faster at 1,024 and 13x at 4,096 (Table 4).

Fused decode kernels. For scalar 4-bit projections, q, k and v run as one GEMV; gate and up as one GEMV with SiLU(gate) $\cdot$ up applied in the reduce; the residual add happens inside the o and down reduces; and RoPE and the KV-cache append share one launch. By the code's count about 28 launches per layer become 13 (no profiler was available on the pods); `TRAPETUM_FUSE=0` restores the separate kernels. Prompt logits are byte-identical to the separate kernels (checked on an RTX 4090 and on an A40), greedy matches 16/16 in every run, and a grouped-query model with head dimension 64 matches a CPU reference (relative error 7.9×10^{-4}, top-1 5/5). Over the first 128 positions the 4-bit decode goes from 6.36 to 5.28 ms per token on an RTX 4090 (17% less time, 157 to 189 tok/s) and from 10.49 to 9.31 ms on an A40. A first version that read the projections from a table passed by value was about 1.8 ms per token slower on an A40; passing them as scalar arguments made it fast. The 2-bit path gets only the RoPE and KV-append fusion, as its additive-codebook projections are not fused (about 1.5% on an A40, 7.17 to 7.06 ms); its 2x8 decode sums partials with atomics, so its logits vary slightly from run to run, and the bit-identity results apply to the 4-bit path.

Tensor-core 4-bit GEMV. The 4-bit codebook GEMV now runs on tensor cores (`mma.sync` m16n8k16, f16 inputs, f32 accumulation), reading the weights straight from the 4-bit index bytes

with no repacking or extra memory and adding partial sums in a fixed order; decode and the batch-invariant verify share the kernel. Decode goes from 5.42 to 5.14 ms per token on the 512-token benchmark (184.3 to 194.5 tok/s, +5.5%) and, over positions 6 to 22 on the same host, from 184.6 to 205.0 tok/s with CUDA graphs and from 175.8 to 195.4 tok/s eager. Greedy decoding still matches HuggingFace 16/16, with a worst prompt-logits error of 1.94×10^{-3} (1.98×10^{-3} before). The kernel needs `sm_80` or newer and was measured on an RTX 4090 only; older GPUs, `TRAPETUM_TC=0`, the non-default determinism modes and the sparse path keep the previous FMA kernels and their bits (with `TRAPETUM_TC=0` and on the sparse path, prompt logits are byte-identical to the previous commit), and the 8-bit LM head stays on FMA kernels.

Together. At short context (first 128 positions) the 4-bit decode took 7.63 to 7.71 ms per token (130 to 131 tok/s) before split-KV and 5.28 to 5.44 ms (184 to 189 tok/s) after split-KV and fusion, on different hosts: +40% to +46% depending on the host pair. At 1,024 positions, 34.3 to 35.0 tok/s became 170.9 tok/s, about 5x across hosts. At 4,096 positions only split-KV has been measured (10x); nothing was measured at that length after fusion or the tensor-core GEMV. Apple Metal is unchanged: it keeps the sequential attention, runs the same separate operations as before and refuses sparsity; split-KV, the fused kernels, sparsity and the tensor-core GEMV are CUDA only for now.

Opt-in activation sparsity. A threshold file named by `TRAPETUM_SPARSITY` gives per-layer thresholds (TEAL-style [15]): inputs of the q/k/v, o, gate/up and down projections below their threshold become zero, and the 4-bit GEMVs skip the index rows of zero inputs. It is off by default, lossy and CUDA only; without a file the kernels and their bits are unchanged, and Metal and the batched speculative path refuse it. Position 0, the attention sink, always runs dense: sparsifying it as well cost +0.66 PPL at 25% sparsity, against +0.12 with it kept dense. At 25% on every site, decode is 1.16x faster over the first 128 positions (5.44 to 4.69 ms per token, 184 to 213 tok/s) and 1.15x at 1,024 positions (5.85 to 5.08 ms), for +0.12 PPL (6.10 to 6.22) and +0.11 on the full test set (5.72 to 5.83), computed in Python with the file's weights. The runtime's own NLL on the first 2,048 wikitext-2 test tokens reproduces the simulated loss (+0.0174 against +0.0164 nats per token), and the two agree within 2.4×10^{-5} nats when dense. Per-site mixes with the o and down inputs at 50 to 60% reach 1.19 to 1.26x but cost +0.17 to +0.30 PPL. Sparse decoding changes the output: at 25% the 16-token greedy continuation matches the dense reference for 3 of 16 tokens.

12.3 Batch-invariant verification and PARD speculative decoding

Verify. A draft of $M = 1$, 2, 4, 8 or 16 tokens is verified in one forward whose logits are, row by row, the same bits as M decode steps, so a speculative decoder built on it emits exactly what plain greedy decoding emits. 15 of 15 checks pass ($M = 1$, 2, 4, 8 and 16 at bases 6, 56 and 1,016, attention over up to 17 chunks, CUDA graphs replayed at other bases, state scrambled before every checked call, next decode step compared), also with `TRAPETUM_FUSE=0` and with `TRAPETUM_TC=0`. At short context the verify costs 1.05x a decode step at $M = 2$, 1.06x at $M = 4$, 1.12x at $M = 8$ and 1.21x at $M = 16$; the cost grows with context, to 1.46x at $M = 16$ near 1,100 positions. An FMA-only verify, kept as the fallback, reached the same bits but cost 3.27x a decode step at $M = 16$ (3.10x after a second attempt), in two earlier runs on other hosts. Prompts are prefilled through the same verify graphs (chunks of 16, then 8, 4, 2 and 1 rows), whose rows and KV-cache rows carry the decode step's bits, so a prefilled prompt continues exactly like a decoded one; plain and speculative runs share this prefill.

PARD. PARD [13, 14] drafting, in target-independent mode, now runs in the Rust runtime: the drafter proposes K tokens in one forward over the new real tokens plus $K-1$ mask tokens, the target checks the last token and the K drafts in one batch-invariant verify, and the longest

matching prefix plus the target's next token are emitted. Both caches are addressed by position, so a rejection needs no copy. In the decision run the target is Llama-3.1-8B-Instruct in 4-bit (new exporter defaults, 5,041 MiB), the drafter `amd/PARD2-Llama-3.1-8B` (1.2 billion parameters, 16 layers) in 4-bit with a 4-bit head, $K = 7$ (the target verifies 8 rows), with at most 256 new tokens and EOS stop, on an RTX 4090 with CUDA graphs in both arms. The 260 prompts (80 GSM8K, 80 HumanEval and 20 MATH-500 from PARD's repository, 80 MT-Bench first turns from FastChat) use the chat template with a system prompt and are up to 437 tokens long; each ran twice with the arm order alternated, both arms on the same file in the same process. All 520 speculative outputs are identical, token for token up to EOS or 256 tokens, to plain greedy decoding of the same 4-bit target. End to end with prefill included (total new tokens over total wall time from prefill start to last token), throughput goes from 165.5 to 415.6 tok/s, 2.51x pooled over 520 runs and 108,102 tokens, with 4.19 tokens emitted per verify round, the bonus token included (per prompt set in Table 4). With an 8-bit drafter head (one repetition) the gain is 2.31x (165.4 to 381.9 tok/s, 4.22 tokens per round), 260 of 260 outputs identical; that drafter's 16-row forward costs 0.43 of a target decode step, against 0.27 with the 4-bit head.

On this grouped-query target, the tensor-core path gives prompt logits within 1.93×10^{-3} relative error of the HuggingFace reference (top-1 6/6) and decodes at 173.5 tok/s (5.76 ms per token, 512-token benchmark); its 16-token greedy continuation follows the reference for 4 tokens, then splits on a near-tie (top-two logit margin 0.004 at the fifth token). Its verify passes 15 of 15 bit checks; 8 tokens cost 1.19x a decode step and 16 tokens 1.34x to 1.36x at short context, rising to 1.58x near 1,050 to 1,170 positions. MT-Bench gains least: in a preliminary acceptance check with PARD's own code, its acceptance came out 22.5% below the PARD-2 paper's vLLM figure, while GSM8K, MATH-500 and HumanEval were within 5%. The July two-model pair on Llama-3.1 (a Llama-3.2-1B drafter) stayed at parity (0.85x to 0.90x), but in another harness on the previous runtime, so the two results are not a controlled A/B.

12.4 Llama-2 70B in 2 bits on one RTX 4090

On 7 November 2026 the 2-bit path of Section 7 served the AQLM 2x8 checkpoint `ISTA-DASLab/Llama-2-70b-AQLM-2Bit-2x8-hf` (not PV-tuned) on a single RTX 4090. The exporter copies the AQLM codes, codebooks and scales bit for bit and quantizes the fp16 LM head to 8 bits, a 17.96 GB file; the runtime keeps the embedding off the GPU (524 MB in fp16 in the file, about 1.05 GB as f32 in host RAM, one row uploaded per token). Llama-2 70B uses the Llama-2 7B tokenizer, so the perplexities compare directly (Table 5). This is one run on one host, against a rule fixed before the run, whose five criteria all passed: the model fits on the card; prompt logits within 5×10^{-3} relative error of the reference, top-1 on every prompt position and greedy 16/16 in eager and CUDA-graph mode; the runtime's own NLL within 0.005 nats of the reference; full-test perplexity at most 4.70; at least 10 tok/s.

Llama-2	file	PPL, 30 windows	PPL, full test	decode tok/s
7B, fp16	13.48 GB	5.83	5.47	
7B, 4-bit, new exporter	3.75 GB	6.10	5.72	205.0
70B, 2-bit AQLM 2x8	17.96 GB	**4.68**	**4.49**	38.4

Table 5: Llama-2 70B in 2 bits against Llama-2 7B. Wikitext-2 at sequence length 2048, computed in Python on the served weights; decode at batch 1 with CUDA graphs on an RTX 4090, positions 6 to 22, the two models on different hosts.

Against the Hugging Face AQLM model carrying the same 8-bit head, prompt logits are within 1.77×10^{-3} to 2.57×10^{-3} relative error (top-1 6/6) and the 16-token greedy continuation matches, in eager mode and in three CUDA-graph runs, with a smallest top-two logit margin of 0.17 along it. On the first 2,048 wikitext-2 test tokens the runtime's own mean NLL is 1.228539 nats per token against 1.228504 for the reference. With a 1,024-position KV cache, decoding peaks at 17,378 MiB of the card's 24,564 MiB (17,698 MiB with the 1,216-position cache of the decode to 1,152 positions, 18,004 MiB with the 2,112-position cache of the 2,048-token NLL check). Decode runs at 38.3 and 38.6 tok/s at positions 6 to 22, 38.1 over a 256-token decode and 37.5 near 1,024 to 1,152 positions: about 17.4 GB of weights are read per token, so about 670 GB/s, two thirds of the card's bandwidth. On the same host the 7B 2-bit PV file decodes at 213.4 and 213.9 tok/s. The checkpoint is 2x8 and not PV-tuned (no PV-tuned 2x8 Llama-2 70B is published): at context 4,096 the model cards give 4.21 for it, 3.83 for the 1x16 checkpoint and 3.78 for the PV-tuned 1x16 one, which the runtime now also serves (Section 12.5); the 4.06 of Table 3 is that 1x16 checkpoint (not PV-tuned), run in Python on an H100, not this file.

12.5 Llama-2 70B PV-tuned in 1x16, with exact speculative decoding

The PV-tuned [5] Llama-2 70B is published only in the 1x16 layout of AQLM [4], `ISTA-DASLab/Llama-2-70b-AQLM-PV-2Bit-1x16-hf`: each weight matrix has one codebook of 65,536 entries of 8 fp16 values and one 16-bit code per group of 8 weights, still 2 bits per weight. The exporter copies these codes, codebooks and scales bit for bit too (8-bit LM head as in Section 12.4), an 18.54 GB file. A 1 MiB codebook does not fit in shared memory, so, like AQLM's own 1x16 CUDA kernel, the decode kernel gives each output row to one warp and gathers every codebook entry from the L2 cache; it computes the products in f32 rather than fp16 and uses no atomics, so its results are the same bits from run to run, eagerly and in CUDA graphs. Measured on 7 November 2026 on one RTX 4090, in two runs, each against a rule fixed before it (Table 6).

Llama-2, 2-bit AQLM	file	PPL, 30 windows	PPL, full test	decode tok/s
13B PV, 1x16	4.00 GB	5.74	5.40	42.6
70B, 2x8 (Section 12.4)	17.96 GB	4.68	4.49	38.4
70B PV, 1x16	18.54 GB	**4.17**	**4.02**	8.5
70B PV, 1x16, speculative ($K = 7$)	+ 2.05 GB	**4.17**	**4.02**	**22.5**

Table 6: PV-tuned 1x16 checkpoints served by the runtime on an RTX 4090. Wikitext-2 at sequence length 2048, computed in Python on the served weights (the new 7B 4-bit file: 6.10 and 5.72). Decode at batch 1 with CUDA graphs, positions 6 to 22, except the speculative row: end-to-end throughput, prefill included, over 12 prompts of 8 to 65 tokens with up to 128 generated tokens each and a 2-bit Llama-2 7B drafter (2.05 GB), against 8.1 tok/s for plain decoding in the same harness. The 2x8 row comes from another host.

Prompt logits are within 1.62×10^{-3} relative error of the Hugging Face AQLM model on its dequantize-then-matmul path (top-1 6/6), and the 16-token greedy continuation matches in eager and CUDA-graph mode. Against its default path the gap is 5.91×10^{-3}: aqlm 1.1.7 runs a prompt of 6 tokens or fewer through its GEMV kernel, which adds the 8 products of each group in fp16, and that path is itself 4.96×10^{-3} away from the dequantized one. On the first 2,048 wikitext-2 test tokens the runtime's mean NLL is 1.132485 nats per token against 1.132619 for the reference (13B: 1.362944 against 1.362946). With a 1,024-position KV cache the 70B peaks at 17,930 MiB (18,560 MiB with the 2,112-position cache of the NLL check). Plain decode runs at 8.5 tok/s at positions 6 to 22 and at 8.4 up to 1,152 positions: every group of 8 weights costs a random 16-byte codebook read that L2 serves as a 32-byte sector, and counting the whole decode step the 7B PV

1x16 and the 70B move about 2.0 and 2.3 TB/s of such sectors. On the same host the 7B PV 1x16 decodes at 78 tok/s against 215 for its 2x8 version.

Speculative decoding [12] reuses each codebook read for several tokens. The 2-bit 2x8 Llama-2 7B (2.05 GB, same tokenizer) drafts K tokens with CUDA-graph decode steps, and the 70B verifies the $K+1$ rows in one pass with a batch-invariant kernel: each row gets the decode kernel's chain of fused multiply-adds and its reduction order, so the same bits, while each code and codebook entry is read once for all rows. At short context a verify pass of 8 rows costs 1.11 decode steps and one of 16 rows 1.81 (1.14 and 1.87 near 1,050 to 1,070 positions), and the verify rows are identical to decode steps at three bases, for $M = 1$, 2, 4, 8 and 16, eager and CUDA graph. Over 12 prompts (8 windows of 64 wikitext-2 test tokens and 4 short prompts, BOS included), 128 greedy tokens each at most, the outputs equal plain greedy decoding in 36 of 36 runs ($K = 3$, 7 and 15). The throughput, end to end from the start of the prefill to the last token (the drafter's prefill, one decode step per prompt token, included), is 22.5 tok/s at $K = 7$ against 8.1 for plain decoding in the same harness (2.79x, 4.49 tokens per verify pass), 18.3 at $K = 3$ and 17.1 at $K = 15$; with both models and the verify buffers, memory peaks at 21,295 MiB. The first run met its criteria of fit, top-1, greedy, NLL agreement and quality (full-test perplexity at most 4.20) but not its speed criterion (at least 10 tok/s), and its logits criterion (5×10^{-3}) failed against the default reference only. The second run met all four of its criteria: identical outputs, verify rows identical to decode steps, at least 12 tok/s at $K = 7$, and logits within 5×10^{-3} of the dequantized reference.

12.6 Llama-3.1 70B Instruct on one RTX 4090, with exact speculative decoding

On 8 November 2026 the 1x16 path served the conversational `ISTA-DASLab/Meta-Llama-3.1-70B-Instruct-AQLM-PV-2Bit-1x16` (PV-tuned, vocabulary of 128,256 tokens), with the 2x8 `ISTA-DASLab/Meta-Llama-3.1-8B-Instruct-AQLM-PV-2Bit-2x8-hf` (3.46 GB file) as the drafter; the 70B file is 21.00 GB with the 8-bit LM head. The 70B checkpoint's current configuration declares no RoPE scaling (8,192 positions), unlike Meta's Llama 3.1 and the checkpoint's own first revision on Hugging Face (llama3 scaling, 131,072 positions); it is served as published: the runtime, like Hugging Face, uses plain RoPE with $\theta = 500{,}000$. On the 24 GB card the export keeps the 2.1 GB input embedding on the CPU, computes the perplexity loss over chunks of positions, frees the dense LM head before loading its dequantized copy and transposes the head on the CPU for its quantization; its highest memory sample (every 300 ms, with `PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True`) was 23,091 MiB of 24,564. One run on one host, against a rule fixed before it, whose seven criteria all passed: fit, logits within 5×10^{-3} of the dequantized-path reference with greedy 16/16, runtime NLL within 0.005 nats of the reference, full-test perplexity at least 15% below the drafter's, outputs identical to plain greedy decoding, verify rows identical to decode steps, and at least 12 tok/s at $K = 7$ (Table 7).

Prompt logits are within 1.40×10^{-3} relative error of the Hugging Face AQLM model on its dequantize-then-matmul path (1.61×10^{-3} against its default path; the two paths are 2.14×10^{-3} apart), top-1 6/6, and the 16-token greedy continuation matches in eager mode and in two CUDA-graph runs. On the first 2,048 wikitext-2 test tokens the runtime's mean NLL is 1.593123 nats per token against 1.593003 for the reference. Memory peaks at 18,730 MiB with a 1,024-position KV cache (19,360 MiB with the 2,112-position cache of the NLL check) and at 21,774 MiB with the drafter and the verify buffers. Plain decode runs at 8.7 tok/s. Over 16 chat prompts (the first 4 of PARD's gsm8k, humaneval and math_500 sets and of the MT-Bench first turns, with the Llama 3.1 chat template and the system prompt "You are a helpful assistant."), the speculative outputs equal plain greedy decoding in 32 of 32 runs ($K = 3$ and 7), and the end-to-end throughput is 21.3 tok/s at $K = 7$ against 7.9 for plain decoding in the same harness (2.7x, 5.59 tokens per verify pass), and

Llama 3.1 Instruct	file	PPL, 30 windows	PPL, full test	decode tok/s
8B PV, 2x8 (drafter)	3.46 GB	11.28	10.61	183.2
70B PV, 1x16	21.00 GB	**6.62**	**6.28**	8.7
70B PV, 1x16, speculative ($K = 7$)	+ 3.46 GB	**6.62**	**6.28**	**21.3**

Table 7: Llama-3.1 Instruct served by the runtime on an RTX 4090. Wikitext-2 with the Llama 3 tokenizer (141 windows of 2,048 tokens), computed in Python on the served weights; these perplexities do not compare with the Llama 2 ones, and no unquantized model was measured. Decode at batch 1 with CUDA graphs, positions 6 to 22, except the speculative row: end-to-end throughput, prefill included, over 16 chat prompts of 53 to 178 tokens with up to 128 generated tokens each, against 7.9 tok/s for plain decoding in the same harness.

17.0 at $K = 3$; at $K = 7$, 22.75 tok/s on gsm8k, 20.87 on humaneval, 23.88 on math_500 and 18.67 on MT-Bench (4.10 tokens per pass). At short context a verify pass of 8 rows costs 1.13 decode steps and one of 16 rows 1.85 (1.16 and 1.92 near 1,050 to 1,070 positions), and the verify rows are identical to decode steps at three bases, for $M = 1$, 2, 4, 8 and 16, eager and CUDA graph. The drafter alone decodes at 183.2 tok/s.

13 Limitations and scope

This is a kernel and benchmark study, not a new quantization method. The scheme is a scalar per output channel codebook (SqueezeLLM family), which trails vector and trellis methods (QTIP [7], AQLM [4], QuIP# [6]) on low-bit accuracy. The accuracy harness uses the HuggingFace forward path, not a tuned serving engine, so absolute throughput is not a serving claim. Our headline kernel baseline is cuBLAS fp16; the comparison against specialized peers (Marlin [8], Machete, FLUTE [10], VQ-LLM [11]) is only partial, as detailed in the baselines section: we decode AQLM's real 2-bit weights faster than fp16, but the op-level timing against AQLM's own kernel is unfinished and we make no matched-accuracy speed claim against Marlin. Prefill stays at about 0.21x cuBLAS, compute bound as expected. Numbers were taken at a single 4096×4096 shape and will differ at other shapes. The microbenchmarks also use random indices and codebooks; real k-means indices and weight distributions may shift cache and bank-conflict behavior, which we have not isolated from the synthetic case.

The November 2026 results (Section 12) have their own limits. The new decode kernels are CUDA only and Metal is unchanged; the tensor-core GEMV needs `sm_80` or newer and ran on an RTX 4090 only. Activation sparsity is lossy and opt-in. Verify cost grows with context (at $M = 16$, on Llama-2 7B, 1.21x a decode step at short context and 1.46x near 1,100 positions; on the Llama-2 70B 1x16, 1.81x at short context and 1.87x near 1,050 to 1,070 positions, and on Llama-3.1 70B Instruct, 1.85x at short context and 1.92x near 1,050 to 1,070 positions). PARD was measured on four prompt sets only, with a 4-bit target and a 4-bit drafter, in target-independent mode, at contexts under about 700 positions (prompts up to 437 tokens plus at most 256 new tokens), and its exactness is relative to plain greedy decoding of that 4-bit target. Neither the Llama-3.1 weights (Llama 3.1 Community License) nor the PARD-2 drafter is redistributed. The new exporter's perplexity was measured on Llama-2 7B only, no perplexity was measured for the new 4-bit Llama-3.1-8B file (the PARD target), and the 2-bit and 4-bit paths have not been compared on one host since the November kernels. The Llama-2 70B 2x8 result is one run on one host, with a 2x8 checkpoint that is not PV-tuned; its perplexities are computed in Python on the served weights, the runtime's NLL was compared with them on the first window only, the 2x8 path has no

exact speculative decoding (the 1x16 path has one), and the Llama 2 weights are not redistributed (Llama 2 Community License). The 1x16 results of Section 12.5 are one run per result on one host each, with greedy decoding only; plain 1x16 decode failed its pre-registered speed criterion of 10 tok/s, and the speculative throughput depends on how many drafts are accepted and on the prompts (at $K = 7$, 22.16 tok/s on the wikitext windows, with 4.68 tokens per verify pass, and 23.15 on the short prompts, with 4.13; the wikitext prompts are 65 tokens long and their prefill is timed). The Llama-3.1 70B Instruct result of Section 12.6 is one run on one host, with greedy decoding on 16 chat prompts; its perplexities use the Llama 3 tokenizer and no unquantized model was measured, the plain RoPE comes from the checkpoint's current configuration (Meta's llama3 scaling was not tested with these weights), and its export leaves at most 1,473 MiB of the card free (highest 300 ms sample).

14 Reproducibility and code

All code is public as **Trapetum**, a single repository that builds and runs with one command and captures GPU, driver, CUDA version, and seed in its outputs. The CUDA kernels are in `kernels/`, the model-level scripts in `model/`, the quantization benchmark in `bench/`, the pure-Rust runtime in `runtime/`, and this paper in `paper/`. The changes of Section 12 are commits 2e8c170 to 6b8cb42 on its main branch; their results are summarized in that section.

https://github.com/neuralboot/trapetum
https://neuralboot.com/trapetum

15 Conclusion

At batch one, decode is bandwidth bound and the arithmetic units are nearly idle. A fused 4-bit codebook GEMV that reads one quarter of the weight bytes turns that idle headroom into a 2.2 to 2.34x decode speedup on the GPUs most people run, while tying cuBLAS on bandwidth rich datacenter parts. Captured as a CUDA graph the way serving engines run, this becomes a 2.0x (7B) to 2.45x (13B) end-to-end decode speedup on real models, where a naive per-layer integration instead loses. Extending the kernel to additive vector codebooks, the trained scheme beats the unweighted scalar codebook at equal bits and equal speed, a step toward fast and accurate low-bit decode. The complementary quantization benchmark shows the same theme from the model side: low-bit quantization is a memory lever first, and on large models it is what makes the model runnable at all, a 2-bit 70B on a single consumer card. The honest negative results map where the easy speedups are not. The November 2026 revision corrects the accuracy of the shipped 4-bit Llama-2 7B file (6.79; the new exporter's file scores 6.10) and carries the same bandwidth logic into the engine: split-KV attention removes most of the decode slowdown with context, and a batch-invariant tensor-core verify lets PARD speculative decoding run Llama-3.1-8B-Instruct 2.51x faster end to end, with output identical to plain greedy decoding of the same 4-bit model; the same engine serves a 2-bit Llama-2 70B on one RTX 4090 at 38.4 tok/s, with a full-test perplexity of 4.49 against 5.72 for the new 7B 4-bit file (4.68 against 6.10 on 30 windows), and its PV-tuned 1x16 version at a perplexity of 4.02, at 22.5 tok/s with exact speculative decoding, as well as Llama-3.1 70B Instruct at 21.3 tok/s on chat prompts with a 2-bit Llama-3.1 8B Instruct drafter.

References

[1] J. Zhao, M. Wang, M. Zhang, Y. Shang, X. Liu, Y. Wang, M. Zhang, and L. Nie. Benchmarking Post-Training Quantization in LLMs: Comprehensive Taxonomy, Unified Evaluation, and Comparative Analysis. arXiv:2502.13178, 2025.

[2] S. Kim, C. Hooper, A. Gholami, et al. SqueezeLLM: Dense-and-Sparse Quantization. arXiv:2306.07629, 2023.

[3] J. Lin, J. Tang, H. Tang, S. Yang, et al. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. Proc. MLSys, 2024. arXiv:2306.00978.

[4] V. Egiazarian, A. Panferov, D. Kuznedelev, E. Frantar, A. Babenko, and D. Alistarh. Extreme Compression of Large Language Models via Additive Quantization. Proc. ICML, 2024. arXiv:2401.06118.

[5] V. Malinovskii, D. Mazur, I. Ilin, D. Kuznedelev, K. Burlachenko, K. Yi, D. Alistarh, and P. Richtarik. PV-Tuning: Beyond Straight-Through Estimation for Extreme LLM Compression. Proc. NeurIPS, 2024. arXiv:2405.14852.

[6] A. Tseng, J. Chee, Q. Sun, V. Kuleshov, and C. De Sa. QuIP#: Even Better LLM Quantization with Hadamard Incoherence and Lattice Codebooks. Proc. ICML, 2024. arXiv:2402.04396.

[7] A. Tseng, Q. Sun, D. Hou, and C. De Sa. QTIP: Quantization with Trellises and Incoherence Processing. Proc. NeurIPS, 2024. arXiv:2406.11235.

[8] E. Frantar, R. L. Castro, J. Chen, T. Hoefler, and D. Alistarh. MARLIN: Mixed-Precision Auto-Regressive Parallel Inference on Large Language Models. arXiv:2408.11743, 2024.

[9] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. Proc. ICLR, 2023. arXiv:2210.17323.

[10] H. Guo et al. Fast Matrix Multiplications for Lookup Table-Quantized LLMs. Findings of EMNLP, 2024. arXiv:2407.10960.

[11] VQ-LLM: High-Performance Code Generation for Vector Quantization Augmented LLM Inference. arXiv:2503.02236, 2025.

[12] Y. Leviathan, M. Kalman, and Y. Matias. Fast Inference from Transformers via Speculative Decoding. ICML, 2023. arXiv:2211.17192.

[13] Z. An, H. Bai, Z. Liu, D. Li, and E. Barsoum. PARD: Accelerating LLM Inference with Low-Cost PARallel Draft Model Adaptation. arXiv:2504.18583, 2025.

[14] Z. An, T. Liu, Z. Liu, D. Li, R. Liu, and E. Barsoum. PARD-2: Target-Aligned Parallel Draft Model for Dual-Mode Speculative Decoding. arXiv:2605.08632, 2026.

[15] J. Liu, P. Ponnusamy, T. Cai, H. Guo, Y. Kim, and B. Athiwaratkun. Training-Free Activation Sparsity in Large Language Models. Proc. ICLR, 2025. arXiv:2408.14690.

[16] K. Kamahori et al. Fiddler: CPU-GPU Orchestration for Fast Inference of Mixture-of-Experts Models. ICLR, 2025. arXiv:2402.07033.

[17] KTransformers: Unleashing the Full Potential of CPU/GPU Hybrid Inference for MoE Models. Proc. SOSP, 2025.

[18] MegaScale-Infer: Serving Mixture-of-Experts at Scale with Disaggregated Expert Parallelism. Proc. ACM SIGCOMM, 2025. arXiv:2504.02263.

[19] PIPO: Pipelined Offloading for Efficient Inference on Consumer Devices. arXiv:2504.03664, 2025.

[20] HOBBIT: A Mixed Precision Expert Offloading System for Fast MoE Inference. arXiv:2411.01433, 2024.

[21] DAOP: Data-Aware Offloading and Predictive Pre-Calculation for Efficient MoE Inference. Proc. DATE, 2025. arXiv:2501.10375.

[22] PreScope: Unleashing the Power of Prefetching for Resource-Constrained MoE Inference. arXiv:2509.23638, 2025.

[23] MoE-Lightning: High-Throughput MoE Inference on Memory-Constrained GPUs. Proc. ASPLOS, 2025. arXiv:2411.11217.

[24] MoE-Lens: Towards the Hardware Limit of High-Throughput MoE LLM Serving Under Resource Constraints. arXiv:2504.09345, 2025.