Native ROCm Inference on a Consumer RDNA3.5 APU: Reaching llama.cpp Decode Parity via a Unified 1-bit-to-Full Quantization Compute Core

Zach Kelling

Hanzo AI

June 2026

Abstract

We report a measurement-driven engineering campaign that took native-ROCm large language model inference on a consumer RDNA3.5 APU—the AMD Ryzen AI Max+ 395 "Strix Halo" (Radeon 8060S iGPU, `gfx1151`), native Windows, ROCm 7.1, single GPU—from the slowest of the project's five backends to effective decode parity with `llama.cpp` on the *same* silicon. Starting from an f32 path that could not even run quantized models (prefill 39 tok/s = 0.033× llama; decode 1.7 tok/s = 0.068×), the campaign reached decode 23.0–24.5 tok/s (0.92–0.98×) and prefill 941–1015 tok/s (0.80–0.86×), while keeping every quantization format bit-exact against the CPU reference. Two structural ideas carried the work: (i) a *unified* compute core that serves the entire 1-bit-to-8-bit quantization zoo from one templated GPU kernel per stage plus one decode function per format, reused across decode, prefill, and mixture-of-experts; and (ii) relentless roofline discipline—profiling the *production* kernel, attributing every millisecond, and optimizing only what the profiler proved to be the bottleneck. The decisive decode win was a lane-strided block iteration that lifted the matvec from 103.5 GB/s (45% of peak) to 245.8 GB/s (97% of peak, a 2.37× gain), placing the decoder at the memory-bandwidth wall. We formalize this ceiling as a roofline theorem, prove a bit-exactness lemma for the unified core, and prove that the lane-strided decode attains $\geq 97\%$ of peak bandwidth. We are explicit about what was *not* achieved: we do not beat llama.cpp here, and the residual prefill gap is one well-scoped multi-wave WMMA flash-attention kernel.

1 Introduction

A consumer APU is a brutally honest target for inference engineering. The Radeon 8060S has no discrete VRAM—it shares LPDDR5X with the CPU—and a measured streaming-read bandwidth of 231 GB/s (device-to-device copy 207, stream-copy 208; the spec sheet claims 256). It exposes RDNA3.5 matrix cores via the `__builtin_amdgcn_wmma_*` intrinsics of the ROCm stack [7], but on this machine neither `rocwmma` nor MIOpen is installed, and the WSL `hipGetDeviceProperties` call even *misreports* the compute-unit count (20, the WGP count; `rocminfo` confirms 40 CU at 2.9 GHz). It is exactly the environment in which vendor-tuned libraries do not save you and the roofline is close enough to touch.

The thesis of this paper is narrow and measured. Starting from a native-ROCm backend that was the slowest of the project's five—an order of magnitude or worse behind `llama.cpp` (f32 prefill 39 tok/s = 0.033×; decode 1.7 tok/s = 0.068×; even the project's best pre-existing path, the Vulkan backend, reached only ≈ 0.34× decode and ≈ 0.09× prefill)—a kernel-plus-engine campaign reached effective decode parity (0.98×) and ≈ 0.86× prefill on this APU, while keeping every quantization format bit-exact against the CPU reference. We separate, throughout, techniques *ported* from `llama.cpp/ggml` [1, 2] (which are excellent and which we copied deliberately) from genuinely *novel* hanzo IP, and we mark each roadmap item **FEASIBLE-now** versus **ASPIRATIONAL**.

All performance numbers in this paper are measured on that one machine against `llama.cpp` (HIP backend) on the *same* GPU, same model, same quiet-GPU conditions. Where a number is a kernel-isolated microbenchmark rather than an end-to-end run, it is labelled as such. The model used as the primary benchmark is **an 8B dense transformer** [3] (36 layers, $n_{\text{embd}} = 4096$, $n_{\text{ff}} = 12288$, 32 query / 8 KV heads, head dim 128); its resident weights are ≈ 8.04 GB in Q8_0 (≈ 1.06 bytes/element).

Contributions.

- A unified, `WTYPE`-templated compute architecture (`qmatvec_core` for decode, `qmmq_core` for prefill) that serves the full 1-bit-to-full quantization zoo with per-format code reduced to one decode function and one traits row, and the *same* cores driving decode, prefill, and MoE.

- A lane-strided decode kernel that reaches 97% of the memory-bandwidth wall (2.37× over the prior production kernel), establishing effective decode parity with `llama.cpp`.
- A formal roofline analysis: a Decode-Ceiling theorem with a Q8_0 corollary, a bit-exactness lemma for the unified core, and a proposition that the lane-strided decode attains $\geq 97\%$ of peak bandwidth.
- An honest accounting of negative results—a fully-debugged but useless HIP-graph effort, a reverted GQA flash-decode kernel—and a feasibility-marked roadmap for the residual prefill gap.

2 Starting Point and the Shape of the Problem

The matmul shapes that dominate the 8B model are the FFN gate/up ($N = 12288$, $K = 4096$), FFN down ($N = 4096$, $K = 12288$), the four attention projections ($N \in \{1024, 4096\}$, $K = 4096$), and the LM head ($N = 151936$, $K = 4096$). Two regimes matter and they have opposite bottlenecks.

Decode (batch = 1, one new token). Pure streaming: every weight is read once per token and multiplied against a single activation row. This is **memory-bandwidth bound**. The arithmetic floor is trivial; the only question is how close to 231 GB/s the weight reads run. As Section 6 proves, 8.04 GB at the conservative 217 GB/s is ≈ 37 ms, and the ≈ 8.7 GB total resident footprint gives ≈ 40 ms ≈ 25 tok/s—*exactly* the figure `llama.cpp` achieves (tg64 = 25.05), which tells us llama hits the bandwidth wall and 25 tok/s is the physical ceiling, not a software target.

Prefill (process a 512-token prompt). Now activations are a matrix, not a vector; the GEMM is **compute bound** on the matrix cores. `llama.cpp` does pp512 = 1176.3 tok/s here.

The initial native-ROCm state was f32 GEMM with a hard `bail!` on quantized matmul—quantized models could not even run on ROCm. Prefill was 39 tok/s on the f32 path; decode was 1.7 tok/s (a one-block-per-row matvec leaving half the lanes idle). Against llama's 1176/25 that is $\approx 0.033\times$ /$0.068\times$.

3 The Journey, in Measured Steps

3.1 Prefill: f32 → WMMA GEMM → int8 MMQ

The first lever was to stop expanding weights to dense f16. We wrote a native Q8_0 **WMMA GEMM** (`qgemm_q8_0_f16`) that reads the GGML Q8_0 block format directly from VRAM and feeds RDNA3 matrix cores via the raw `__builtin_amdgcn_wmma_f32_16x16x16_f16_w32` intrinsic. A resident dense-f16 weight copy was tried and *rejected*—it slowed decode to ≈ 5 tok/s, because on a shared-memory APU the extra resident copy costs bandwidth you cannot spare. Reading the quant directly is the only viable design here. Prefill progressed f32 39 → Q8_0+WMMA 143 → 32×32 tile 193 → 64×64 double-buffered 278–330 tok/s.

The decisive move was adopting `llama.cpp`'s `mul_mat_q` (MMQ) design: keep weights *int8* and use the integer matrix cores (`__builtin_amdgcn_wmma_i32_16x16x16_iu8_w32`), quantizing *activations* once into a q8 buffer. The single hardest bug in the campaign lived here: the `iu8` builtin's first two boolean arguments are `is_signed` flags, and signed Q8_0 requires them `true`, not `false`—with `false` the inputs read as unsigned, producing huge positive products that NaN across all 36 layers. It was found not by full-model iteration (which only NaNs) but by a dedicated numeric

Table 1: Production `qmatvec_core` (Q8_0 bf16) per-shape throughput before the lane-strided fix.

Shape	**μs/call**	**GB/s**	**% peak**
gate $N12288\,K4096$	394.5	135.6	59
up $N12288\,K4096$	393.4	135.9	59
down $N4096\,K12288$	457.5	117.0	51
lm_head $N151936\,K4096$	4716.3	140.2	61

unit test comparing the GPU kernel against an exact CPU reference from the *same* quantized inputs (`qmmq_numeric.rs`). **Lesson: build the per-kernel numeric oracle first; a NaN in a 36-layer forward is undebuggable.**

MMQ then progressed 170 → 224 (fuse activation-quant into staging) → 340 (llama's actual 128×128 tile) → 473 (a correctness-and-speed insight: our fused in-kernel quant was re-quantizing each activation row ≈ 96×, once per N-column tile of the 12288-wide FFN; llama quantizes activations *once* into a q8_1 buffer, so we de-fused it) → 636 (llama's +4-byte bank-conflict pad, 16-byte vectorized staging, shared-staged scales) → 651 stable. **Lesson: don't blind-tune—copy llama's actual config.** Two ISA-backed counter-examples to naive porting: llama's literal 8-warp / 8-fragment layout *regresses* here (≈ 425 versus 474 tok/s)—with our hand-rolled `v4i` fragments it spills (192 VGPR + ≈ 541 spills, 7 waves) versus the 16-warp / 4-fragment layout (132 VGPR, 10 waves, 0 spills); and stream-K is not llama's RDNA3 path (disabled for non-CDNA AMD), so we did not build it.

The disassembly's honest conclusion: *we are more occupancy-efficient than llama here* (124 VGPR / 10 waves versus llama's measured 168–240). The residual prefill-GEMM gap is therefore not registers or occupancy—it is **weight-load coalescing**. GGML Q8_0 interleaves a 2-byte f16 scale every 34 bytes, so weight quants land at byte $34 \cdot \text{blk} + 2$ (only 2-byte aligned) and global loads cannot go wide. A weight **repack** to contiguous int8 plus separate f16 scales (llama's `block_q8_1_mmq`) fixes it—*proven* at 739 tok/s on a 4B dense transformer at Q8_0—but on 8B it doubles resident weight VRAM (raw Q8_0 for decode *and* the repacked arrays for prefill ≈ 17 GB), silently spills on the 8060S budget, and *collapses* prefill to 58 tok/s. Honest partial result: a win *where memory fits*; landing it on 8B means serving *both* paths from the flat repacked arrays (≈ 8.7 GB)—a FEASIBLE-now follow-on, not done.

3.2 Decode: the lane-strided unified core—the parity lever

By mid-campaign a series of engine fusions (Section 4) had taken eager decode to ≈ 20.5 tok/s, and the matvec was *believed* to be near the bandwidth wall. A careful re-profile of the **production** kernel (not a proxy microbench) overturned that belief and produced the single largest decode win.

The subtlety: two matvec kernels existed in the tree and they were *not the same kernel*. The standalone microbench (`bench.hip`) timed a lane-strided loop and clocked ≈ 227 GB/s—fast. But the kernel actually wired into the Rust decode path was the unified `qmatvec_core<WTYPE>`, and in its then-current form it used a **whole-warp-serial-per-block** loop (`for (int b = 0; b < nblocks; ++b)`) with all 32 lanes cooperating on one block, lane l owning position l. Per-shape measurement (`decode_prof.exe`) showed this production kernel running the dominant FFN shapes at only 117–141 GB/s, that is 50–61% of peak (Table 1).

The serial walk could keep only one block's loads in flight (no latency hiding across blocks) and re-loaded the f16 scale on all 32 lanes every block. The fix—**lane-strided block iteration**—is two characters of intent: `for (int b = lane; b < nblocks; b += 32)`, so the 32 lanes issue 32

independent block loads at once, hiding LPDDR5X latency, with a final warp-shuffle reduction. It is bit-faithful for every `WTYPE` (only the iteration pattern changed; per-type decode and warp reduction are preserved).

Result, red-verified: the matvec went 103.5 GB/s (45%) → 245.8 GB/s (97% of peak) = 2.37×; end-to-end eager decode → 23.4 tok/s (0.93× llama, up from 0.82×); and because the core is shared, MoE decode rode the *same* change from 6.34 → 13.6 tok/s (2×). `nbad`= 0 across all six wired quant types, output coherent, no regression. **The matvec is now at the bandwidth wall.**

3.3 The Last Decode Tail, and the Honest Ceiling

With the matvec maxed (≈ 33.7 ms/token, 78% of the token at 97% BW), the remaining gap to 25 tok/s is a ≈ 9.3 ms (22%) non-matvec tail. The instrumented per-op breakdown (greedy, KV span ≈ 68; instrumented ms scaled by 0.68 to recover un-serialized wall time) attributes it: decode attention (`naive_sdpa`) 2.87 ms—the single biggest non-matvec op—then two residual adds 1.60 ms, two RMSNorms 1.48 ms, fused qk-norm+RoPE 1.16 ms, KV-append 0.79 ms, qkv cast 0.80 ms. None dominates.

We built the obvious lever—a fused GQA flash-decode attention kernel to kill the `repeat_kv` cat (8 → 32 heads), the rocBLAS m=1 GEMV regime, the F32 softmax round-trip, and ≈ 5 launches/layer. It is correct (`nbad`= 0, coherent) but it *regressed* decode 24.5 → 21.2 tok/s and was **reverted**. The reason is occupancy: at batch = 1, decode attention launches only 8 blocks across 40 CUs—severe under-occupancy—and rocBLAS' tiny $QK^\top$/AV beats a single-warp kernel for that workload. **Honest verdict: decode attention is not a winnable lever at batch = 1 without split-KV/flash-split-k for occupancy.** With the matvec at the wall and attention not improvable at batch = 1, decode's honest ceiling on this approach is ≈ 24.5 tok/s at short context = 0.98× llama = effectively parity. (Decode does slow with KV span: 23.0 at d4 → 21.0 at d512, the growing attention being the cause—a longer-context regime where a split-KV flash kernel *would* pay.)

3.4 Mixture-of-Experts

MoE rides the same unified core. `RocmDevice::moe_matvec_quant` dispatches each routed expert through `qmatvec_core<WTYPE>` with a resident-bank cache keyed by the CPU bank pointer (working around ROCm's missing `index_add` by writing one output row per slot). A nrows 1 → n NaN bug was found and fixed. MoE decode went 0.93 → 6.34 (resident-bank cache, 6.8×) → 13.6 tok/s (the shared lane-strided fix, 2×). Q4_K and Q8_0 MoE are `nbad`= 0.

4 The Engine Campaign

A pivotal roofline finding reframed the whole effort: *in isolation, our kernels were already at or above llama, but the engine was leaving most of the win on the floor.* The decode matvec alone benchmarked at the memory roofline; the prefill GEMM alone benchmarked *faster* than llama (in-situ ≈ 1414 tok/s versus llama's 1176 total). Yet end-to-end delivered far less. The gap lived in launch overhead and un-fused elementwise ops—at one point ≈ 2312 launches per decode token, most of them tiny 1-row casts, norms, RoPE, and softmax.

The fusion batch that closed it (each red-verified against a numeric oracle, output coherent): **fused RMSNorm** (one block/row, f32 accumulation; decode 7.5 → 12.3 tok/s, ≈ 576 fewer launches/token); **bf16-native matvec** (`qmatvecu_*_bf16`, killing a redundant BF16→F32→F16→BF16 cast chain that ran 756×/token); **fused RoPE** (GPT-NeoX half-rotation, f32 math, ported from

the CUDA RoPE); **fused softmax**; **fused SwiGLU** (silu·mul). Net: decode 11.72 → 20.53 tok/s (+75%), prefill 733 → 937 (+28%), ≈ 1476 launches/token removed—taking the project from ≈ 0.55× to ≈ 0.80× llama on *both* axes purely by removing engine overhead, confirming that none of the gap to that point was silicon.

4.1 The hipGraph Detour—Correct, Exhaustively Debugged, and *Not* the Lever

The most instructive negative result in the campaign is the HIP-graph effort. The premise was textbook: decode was launch-bound (≈ 2312 launches/token), so capture the per-token decode once and replay it as one graph. The project was large—it required first building a ROCm PagedAttention [5] backend (the engine's existing decode-graph machinery is CUDA-only, gated on PagedAttention holding dynamic KV state in stable device buffers), then making three synchronous default-stream ops async so capture would not trip `hipErrorStreamCaptureImplicit`, then chasing a deep coherence bug. Capture succeeded but replay produced a degenerate loop; a 508-step probe found the compute graph *bit-exact* (all 423 forward intermediates, including the 151936-wide LM-head logits, matched eager to `max_abs_diff`= 0.0). The bug was in no kernel: the benchmark model has no sliding window, so paged-attention takes the `use_full` path, and the ROCm graph metadata refreshed only the windowed buffers, cloning the full ones verbatim from warmup, so the captured kernel attended a *frozen span* every token. The fix (mirroring `cuda_graph.rs`) refreshes the full buffers each token; replay became byte-identical to graphs-off over 1568 tokens (same SHA-256).

The punchline: **zero speedup** (graphs-on 12.7 tok/s = graphs-off 12.7). Our own earlier fusion work had already de-launch-bounded decode—there were no launches left to collapse. The graph project ran on a stale premise the fusion had invalidated. **Lesson: re-measure the bottleneck after every major change.** The machinery is correct and dormant (default-off); the ROCm PagedAttention backend it produced is the prerequisite for speculative decode and multi-stream serving, so the detour was not worthless, just not the decode lever it was built to be.

5 The Unified Quant Architecture

The architectural commitment—a user mandate—was: support quantization from 1-bit to full *with no per-quant kernels*. One quant-agnostic GPU core per stage, one decode function per format, reused across decode, prefill, and MoE. Adding a quant must be one decode function plus one traits row plus one table entry, **zero new kernels**. This is realized as three matched pieces, all confirmed in source.

1. Unified decode core `qmatvec_core<WTYPE,XT>`. One templated warp-per-row kernel. Per type it selects a `qdec<WTYPE>::partial<XT>(blk, xb, lane)` device decode (one function body each, bit-exact to the CPU `to_float`) and a `qdw_traits<WTYPE>` row giving {`BYTES, ELEMS, SYMMETRIC, NSC`}. A `DEFINE_QMATVECU(name, WTYPE)` macro emits the f16 and bf16 `extern "C"` entry points. **Six types are wired today** across all the structural classes:

```
DEFINE_QMATVECU(q8_0,  DW_Q8_0)   // sym, 34B/32
DEFINE_QMATVECU(q4_0,  DW_Q4_0)   // sym, 18B/32
DEFINE_QMATVECU(q4k,   DW_Q4_K)   // ASYM, 144B/256
DEFINE_QMATVECU(q6k,   DW_Q6_K)   // signed 6b, 210B/256
DEFINE_QMATVECU(iq4xs, DW_IQ4_XS) // LUT, 136B/256
DEFINE_QMATVECU(tq2_0, DW_TQ2_0)  // ternary, 66B/256
```

The whole zoo collapses onto **two scalar accumulation shapes**, folded inside each decode so the core itself is shape-agnostic: **symmetric** $\text{val}(p) = \text{scale} \cdot q(p)$ (Q8_0/Q4_0/Q6_K/IQ4_XS-via-LUT/TQ2_0) and **asymmetric** $\text{val}(p) = d \cdot sc \cdot q(p) - d_{\min} \cdot m$ (Q4_K/Q5_K, and IQ1 via a $+\delta$ bias).

2. Unified prefill int8 GEMM core `qmmq_core<WTYPE>`. The proven 651-era 128×128 / 16-warp / `iu8`-WMMA / double-buffered Q8_0 GEMM, generalized so the *only* per-type code at prefill is one `decode_w_half<WTYPE>` (plus scale/min reads) and one `wt_traits<WTYPE>` row; the MMA loop and staging are byte-for-byte the proven Q8_0 machine code. Refactoring Q8_0 onto the template produced **byte-identical disassembly** (116 VGPR / 0 spill, same machine code, `nbad`= 0); Q4_0 landed as the first additional native type (`nbad`= 0). Q4_K/Q6_K/IQ/TQ native *prefill* is in flight (those types still dequant-to-f16 at prefill today—an honest caveat, see Section 6).

3. CPU single-source via the `quant_format!` macro. The 11 sub-4-bit IQ/ternary/FP4 formats that previously could not even *load* (the loader had a `bail!` wildcard) were added—load and dequant, bit-exact versus ggml (`nbad`= 0 / `max_err`= 0.0 on deterministic bytes for all 11: IQ2_XXS/XS/S, IQ3_XXS/S, IQ1_S/M, TQ1_0, TQ2_0, NVFP4, Q1_0). Their codebook grids were auto-extracted verbatim from `ggml-common.h` with verified counts. All 11 are declared through a single `quant_format!` macro plus a `for_each_quant!` table that also generates the `mod.rs` plumbing, with byte-exact equivalence to the prior hand-written code proven in `proof.rs`.

Measured, the abstraction is not a tax: the unified Q4_K decode kernel runs ≈ 181 GB/s = 79% of the Q8_0 bf16 matvec's ≈ 229, already bandwidth-efficient despite serving a 4-bit super-block through the same core. The invariant has been proven end to end (tests: `qmatvec_unified_numeric.rs`, `moe_matvec_unified_numeric.rs`).

5.1 Ported versus Novel

PORTED from llama.cpp/ggml (deliberately, not novel): the MMQ int8 matmul *design* (keep weights int8, quantize activations once, integer matrix cores); the 128×128 tile, de-fused activation quant, bank-conflict pad, 16-byte vectorized staging; the GGML block formats and bit-exact dequant semantics; the q8_1-bias accumulation shape for asymmetric types. **GENUINELY NOVEL hanzo IP:** the *unification* itself—`qdec<WTYPE>::partial` + `qdw_traits` + `DEFINE_QMATVECU` and the matched `qmmq_core<WTYPE>` + `decode_w_half<WTYPE>` + `wt_traits`, such that one decode core and one prefill core serve the full 1-bit-to-full zoo with per-format code reduced to a decode function and a traits row, *and the same cores drive decode, prefill, and MoE.* llama.cpp uses per-format `load_tiles_*` front-ends but does not expose a single `WTYPE`-templated core spanning matvec, GEMM, and MoE with a one-line add-a-quant contract. **Honest caveat:** the underlying *math* per format is ggml's; the claimable subject matter is the unifying architecture and generation mechanism, not the dequantization of any individual format.

6 Formal Roofline Analysis

We now make the central claims precise. The roofline model [4] is the spine of this work, and it both certifies that the decoder is finished and bounds what remains.

Definition 1 (Decode step model)**.** *Fix a transformer with total resident weight footprint W bytes that must be streamed once per generated token. Let B be the achieved (sustained) global-memory bandwidth in bytes per second on the production decode path, and let β be the per-token weight*

traffic in bytes, $\beta \geq W$ (equality when no weight is read more than once). The decode throughput in tokens per second is $T = 1/t$, where t is the wall-clock time of one decode step.

Theorem 1 (Roofline Decode Ceiling)**.** *For any memory-bound decoder satisfying Definition 1, the decode throughput obeys*

$$T \;\leq\; \frac{B}{\beta} \;\leq\; \frac{B}{W}.$$

Equivalently, the per-token time is bounded below by $t \geq \beta/B \geq W/B$. A memory-bound decoder is therefore bandwidth-limited: no algorithmic change that preserves the weight footprint and the read-once property can exceed B/W tokens per second.

Proof. One decode step must, by definition of the weight-streaming model, transfer at least β bytes of weights from global memory into the compute units; this is a lower bound because every weight participates in the matvec for the new token and the values are not resident in registers or cache across the step (the working set W exceeds the on-chip cache, which is the defining condition of the memory-bound regime). The hardware moves bytes at a rate no greater than the achieved bandwidth B. Hence the time to perform the mandatory transfers is at least β/B, and since these transfers are on the critical path of the step, $t \geq \beta/B$. Taking reciprocals, $T = 1/t \leq B/\beta$. Finally $\beta \geq W$ gives $B/\beta \leq B/W$, so $T \leq B/W$. Compute time only adds to t and cannot reduce it, so the bound is unconditional for the memory-bound regime. □

Corollary 1 (Q8_0 ceiling on `gfx1151`)**.** *For the 8B model at Q8_0 with total resident footprint $W \approx 8.7$ GB streamed at the conservative achieved bandwidth $B \approx 217$ GB/s, the decode ceiling is*

$$T \leq \frac{217 \text{ GB/s}}{8.7 \text{ GB}} \approx 24.9 \text{ tok/s} \approx 25 \text{ tok/s}.$$

Proof. Substitute $W = 8.7$ GB and $B = 217$ GB/s into Theorem 1: $t \geq 8.7/217 \approx 0.0401$ s, so $T \leq 1/0.0401 \approx 24.9$ tok/s. The bare weight tensor (8.04 GB) at the same B gives the looser ≈ 27.0 tok/s; the 8.7 GB figure includes the activation and KV working set actually streamed per token, which is why ≈ 25 tok/s is the operative ceiling. □

Corollary 1 is exactly the number `llama.cpp` achieves (tg64 = 25.05). This is the paper's most important empirical fact: llama is *also* at the wall, so ≈ 25 tok/s is physics, not a target hanzo is losing to. Our measured decode of 24.5 tok/s is 0.98× this ceiling.

Lemma 1 (Bit-Exactness of the Unified Core)**.** *Let W be a quantized weight matrix in any supported format `WTYPE`, and let $\hat{W} = \mathrm{DEQ}_{cpu}(W)$ be its reference CPU dequantization to `f32`. For any activation vector x, the unified device kernel `qmatvec_core<WTYPE>` computes, for each output row i, a value equal to the reference $\sum_j \hat{W}_{ij}\, x_j$ accumulated in `f32` in the same per-block order—bit for bit, for every supported `WTYPE`.*

Proof sketch. The kernel computes $y_i = \sum_b \big(\sum_{p \in \text{block } b} \mathrm{val}_{\mathrm{dev}}(i, b, p) \cdot x_{j(b,p)} \big)$, where $\mathrm{val}_{\mathrm{dev}}$ is produced by `qdec<WTYPE>::partial`. Three facts compose. (i) *Per-element identity:* for each supported format, `qdec<WTYPE>::partial` is constructed to evaluate the same arithmetic expression as the CPU `to_float`—the symmetric shape $\text{scale} \cdot q(p)$ or the asymmetric shape $d \cdot sc \cdot q(p) - d_{\min} \cdot m$—using the identical `f16`/`f32` scale decode and integer unpack, so $\mathrm{val}_{\mathrm{dev}}(i, b, p) = \hat{W}_{i,j(b,p)}$ as IEEE-754 `f32` bit patterns. (ii) *Order preservation:* both reference and device accumulate block-by-block and, within a block, in increasing position order; the lane-strided variant (Section 3.2) changes only *which lane* issues a block's loads, not the summation tree, because the final warp-shuffle reduction

sums lane partials in fixed lane order, identical to the serial walk's block order. Since `f32` addition is deterministic (though non-associative) and the addition order is fixed and equal on both sides, the running sums coincide at every step. (iii) *Closure under the template:* adding a format introduces only a new `qdec` body and `qdw_traits` row; the accumulation and reduction code is shared and unchanged, so (i)–(ii) hold for the new format by the same argument. The `nbad`= 0 / `max_abs_diff`= 0.0 results of `qmatvec_unified_numeric.rs` across all six wired types are the empirical witness; the refactor producing byte-identical GEMM disassembly witnesses the prefill core analogously. □

Proposition 1 (Lane-strided decode attains $\geq 97\%$ of peak bandwidth)**.** *On `gfx1151`, the lane-strided `qmatvec_core` sustains $\geq 97\%$ of the achievable peak read bandwidth, a $\geq 2.3\times$ improvement over the whole-warp-serial production kernel it replaced.*

Proof. The serial kernel issues, per warp, one block's loads at a time and stalls on LPDDR5X latency L before the next block: with n blocks per row its load-issue critical path is $\approx n\,L$, so utilization is bounded by $\tau/(\tau + L)$ per block where τ is the per-block transfer time; with $\tau \ll L$ on a shared APU this yields the measured 45–61% (Table 1). The lane-strided kernel issues 32 *independent* block loads simultaneously (lane l takes blocks $l, l{+}32, \ldots$), so up to 32 requests are in flight and the per-block latency L is amortized across them; once the in-flight request count exceeds the latency-bandwidth product $L \cdot B/(\text{request size})$, the pipe saturates and utilization approaches $\tau/(\tau + L/32) \to 1$. Empirically the measured throughput is 245.8 GB/s versus the 103.5 GB/s of the serial kernel, a factor 2.37; against the achievable read peak this is 97% (and exceeds the conservative 231 GB/s stream-copy figure, confirming the read path is faster than the copy path). By Theorem 1 no further matvec speedup is possible without reducing β, so 97% certifies the matvec as finished. □

A complementary hardware finding killed a class of bad ideas: **RDNA3.5 `iu8` WMMA is $\approx$ 1:1 with f16, not** $2\times$ (real sustained ≈ 52 TOPS-i8 / ≈ 49 TFLOPS-f16; the marketed 118/59 are unreachable). So int8's only decode benefit here is the smaller *weight footprint* β (fewer bytes to stream), *not* compute throughput—consistent with Theorem 1, where only β enters.

Prefill is compute-bound. pp512 $\approx$ 1015 tok/s $\Rightarrow\approx$ 504 ms; llama $\approx$ 435 ms. The in-situ GEMM is 70% ($\approx$ 362 ms) at $\approx$ 1414 tok/s—*faster* than llama and not the bottleneck. The largest reducible block is attention materialization at 15% ($\approx$ 79 ms): the `naive_sdpa` path does `repeat_kv` (8 $\to$ 32 head cat), a rocBLAS $QK^\top$ producing a $32\times512\times512$ score tensor per layer, a separate affine-mul for the scale, and an F32 softmax round-trip. The remaining $\approx$ 12% is RoPE/cast/norm/residual glue—already fused, near-irreducible. Prefill parity therefore needs a **multi-wave cooperative WMMA flash-attention prefill kernel** [6] (matrix-core tiled $QK^\top \to$ online softmax $\to$ AV, causal early-exit, no `repeat_kv` cat, no F32 round-trip, no score materialization) to cut attention 79 $\to\approx$ 20–30 ms $\to\approx$ 1150–1200 tok/s $\approx$ parity. A *scalar* flash kernel was tried and lost to rocBLAS (388 versus 563 tok/s) precisely because it used no matrix cores; a *single-wave* WMMA flash matched rocBLAS but did not beat it. The win requires a multi-wave cooperative kernel (4 waves, double-buffered LDS)—a real rewrite, MED-HIGH effort, the largest unrealized prefill lever.

7 Results

End-to-end, eager path, quiet GPU, the 8B model at Q8_0 on `gfx1151`, versus `llama.cpp` HIP on the *same* GPU (llama parity targets: pp512 = 1176.3, tg64 = 25.05). All hanzo configs are

bit-exact (`nbad`= 0) and produce coherent output.

Table 2: End-to-end performance versus `llama.cpp` on the same GPU.

Stage	**Start**	**hanzo**	**llama**	**Ratio**
Decode (d4) tok/s	1.7	23.0–24.5	25.05	0.92–0.98×
Prefill (pp512) tok/s	39	941–1015	1176.3	0.80–0.86×
MoE decode tok/s	0.93	13.6	—	2× prior

Table 3: Kernel-isolated throughput (not end-to-end), same GPU.

Kernel	**Measured**	**Note**
Decode matvec Q8_0 bf16	245.8 GB/s	97% of peak; maxed
Prefill GEMM int8 WMMA	≈1414 tok/s	above llama's 1176
Decode Q4_K matvec	≈181 GB/s	79%; BW-efficient

The matvec figure of 245.8 GB/s was 103.5 GB/s (45%) before the lane-strided fix. Trajectory for context (prefill, tok/s): f32 39 → WMMA 143 → 193 → 278–330 → MMQ 473 → 636/651 → 937–1015. Decode (tok/s): 1.7 → 8.5 (warp-per-row) → fusion 20.5 → lane-strided core 23.4–24.5.

8 Limitations and Future Work

We are explicit about what we did not achieve. We do *not* beat `llama.cpp` on this hardware; the remaining prefill gap (≈ 0.86×) is real engineering work, not a rounding error. The GGUF kernel-plus-fusion approach has an honest ceiling of ≈parity here—decode is at the bandwidth wall (Theorem 1, Proposition 1) and decode attention is not improvable at batch = 1 (the WMMA flash-decode kernel was built, regressed, and reverted). Several formats (Q4_K/Q6_K/IQ/TQ) still dequant-to-f16 at *prefill* today; their decode is already native through the unified core.

FEASIBLE-now. (1) Multi-wave WMMA flash-attention prefill kernel → prefill parity (the #1 lever; the scalar and single-wave attempts and their failure modes are documented; the prerequisite is a `flash.hip` genco fix—include `<hip/hip_bf16.h>` for the WMMA-capable `__hip_bfloat16`, not the legacy `<hip/hip_bfloat16.h>`). (2) Q8_0 repack-for-both-paths: serve decode *and* prefill from the flat repacked int8+f16-scale arrays so the proven 739-tok/s-on-4B coalesced-weight win transfers to 8B without doubling VRAM. (3) Native Q4_K/Q6_K/IQ prefill through `qmmq_core`. (4) Relax the Q4_K k%256 construction guard so FFN widths like 17408 stop falling back to dense f16.

ASPIRATIONAL. (a) Surpassing llama on this APU needs the ROCm-PagedAttention keystone (now built) plus split-KV flash decode for long-context occupancy and speculative decode/MTP. (b) Long-context decode (split-KV / flash-split-k to recover the d512→d1024 falloff). (c) Backend portability via CubeCL: spiked, verdict NOT-YET—CubeCL's HIP compiler does not yet emit the int8 `iu8` intrinsic (it panics on int8) though the hardware has it; viable today for f16/bf16 and Vulkan-int8 only. Keep the hand-tuned HIP int8 path until a tuned CubeCL kernel reaches ≥ 60–70% of it across backends.

9 Method and Lessons

The campaign reduces to a handful of transferable lessons, each earned the hard way. (1) **Measure the real kernel, not a proxy:** the 2.37× decode lever was found only because we profiled the kernel actually wired into the Rust decode path and discovered it was a different, slower kernel than the look-alike microbench. (2) **Roofline first; let the ceiling tell you when to stop:** the matvec is at 97% of the wall, so it is done; llama hitting the same 25 tok/s confirmed 25 is physics. (3) **Copy llama's actual config; don't blind-tune:** reading `mmq.cuh` line by line beat constant-sweeping—and told us which of llama's choices (8-warp, stream-K) to reject on this chip, with ISA evidence. (4) **Build the per-kernel numeric oracle before the kernel:** the `iu8` signedness NaN was undebuggable in a 36-layer forward; a tiny GPU-versus-exact-CPU test isolated it in one shot. (5) **Re-measure the bottleneck after every major change:** the entire hipGraph project was correct, exhaustively debugged, and useless because the fusion work had already removed the launches it was built to collapse. (6) **On a shared-memory APU, bandwidth is the budget:** every resident dense-f16 copy "optimization" backfired (decode → 5 tok/s; the 8B repack spill → 58 tok/s).

The bottom line is deliberately unembellished: on a consumer RDNA3.5 APU with no vendor matmul library, a measurement-driven path took native-ROCm inference from the slowest backend in the project to effective decode parity (0.98×) and ≈ 0.86× prefill against `llama.cpp` on the same silicon, while a unified 1-bit-to-full quant core kept every format bit-exact. We do not beat llama here; the remaining prefill gap is one well-scoped multi-wave WMMA flash-attention kernel, and surpassing llama is a keystone-project (ROCm PagedAttention, now built) away—both stated as engineering, not marketing.

References

[1] G. Gerganov et al. *ggml: Tensor library for machine learning*. 2023.

[2] G. Gerganov et al. *llama.cpp: LLM inference in C/C++*. 2023.

[3] Qwen Team. *Qwen3 Technical Report*. Alibaba, 2025.

[4] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[5] W. Kwon et al. Efficient Memory Management for Large Language Model Serving with PagedAttention. *SOSP*, 2023.

[6] T. Dao et al. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *NeurIPS*, 2022.

[7] AMD. *ROCm: Open Software Platform for GPU Compute*. Version 7.1, 2026.