Preprint · August 2026 · Companion artifacts: llama-cpp-gfx906-turbo (kernel `mmvq-db-gemm.cuh`*), microbenchmarks* `scratchpad/m12-lds-gemm`*, raw measurements BENCHMARKS.md AX–BF*

Why Your Prefetch Isn't Prefetching: Diagnosing Compiler-Defeated Latency Hiding in Small-Batch Quantized GEMM on a Matrix-Core-Less GPU

Sebastian Haas
Independent researcher · `kontakt@sebastianhaas.eu`

***Abstract*—Batched token generation with 13–32 concurrent sequences occupies an awkward regime for quantized large-language-model inference: too wide for the memory-bound matrix–vector kernels that dominate single-stream decoding, too narrow for tile-based GEMM. On a 2018-era AMD Instinct MI50 (gfx906, no matrix cores) this regime showed a throughput *dip*-16 sequences decoded slower than 12-and the existing small-batch kernel sat at 2 waves per SIMD, a factor of 3× above its bandwidth and integer-dot-product floors. We report a diagnosis methodology and a kernel redesign. The methodology combines a *subtraction ladder* (variants that omit staging, LDS reads, or arithmetic to attribute wall time) with a *compressed instruction-order trace* of the compiled ISA. The trace exposed three compiler behaviours that silently reverted every latency-hiding attempt to lock-step execution: loads sunk into conditional blocks, LDS stores hoisted above the compute they were meant to overlap, and arithmetic migrating across memory barriers. Two lines of inline assembly-a memory clobber after the loads and accumulator-binding operands before the stores-restore the intended order. The resulting kernel (double-buffered LDS staging of weights and activations, K-split across waves, deterministic split-K across workgroups, 83 VGPRs, 3 waves/SIMD) runs 1.57× faster than its predecessor across four matrix shapes at fp32 rounding-level error (NMSE < 2×10^{-14}). End to end, 16-sequence decode throughput rises by 44–78 % across six models from 7B to 24B parameters (e.g. Qwen3-8B Q4_0: 380 → 549 tokens/s; Mistral-Small-24B: 126 → 224), 24- and 32-sequence throughput by 20–47 %, with 1–12 sequences untouched. Through a production serving stack the same hardware now exceeds a community vLLM port by 37 % at 16 sequences. Applying the same staging discipline to the prompt-processing GEMM raises prefill throughput by 15 % (pp512: 978 → 1123 tokens/s), and a one-parameter occupancy fix to the long-context attention kernel-tuned for decode but throttled on the prefill-raises 32k-token prefill by 48 %. Routing quantized-KV attention by column count instead of by cache type adds another 37 % at 32k. Against a current upstream build the resulting stack leads by 20–78 % on five production models and by 4× in long-context decode. We close with the observation that the last levers were not instructions but policies, and show that moving speculative decoding under a measured cost regulator-rather than a tuned constant -turns a 23 % loss on prose into a 3 % one. We quantify the remaining gap-nibble unpacking and per-block scaling account for 49 % of vector instructions-and report three negative results with their causes: a K-quant port that gains only 4–5 % against a stronger tiled baseline, a MoE expert-gather variant that is bounded by CPU-offloaded experts, and a bimodal throughput artefact traced to the 2 MiB placement of a scratch buffer.**

***Index Terms*—LLM inference, quantized GEMM, GPU kernel optimisation, latency hiding, AMD GCN, compiler scheduling, llama.cpp**

I. Introduction

Decoding a large language model is a sequence of matrix–vector products: each generated token multiplies the full weight set against one activation vector per sequence. For a single sequence the kernel is memory-bound and well understood. When several sequences are decoded together, the activation side becomes a thin matrix with N columns, and quantized inference engines such as llama.cpp switch between a vector path ($N \le 8$), a hand-tuned small-batch path, and a tile-based integer GEMM ("MMQ") as N grows. The transition points are tuned per hardware generation and rarely revisited.

This paper concerns the regime N = 13–32 on a GPU without matrix cores, the AMD Instinct MI50 (gfx906). Such devices are cheap, plentiful, and-with 16 GB of HBM2 at 1 TB/s-attractive for self-hosted inference, but they lack the MFMA units that all modern small-batch GEMM work assumes. Integer dot products (`v_dot4_i32_i8`) are the only accelerated primitive.

Our starting point was a measurement artefact of a different experiment: comparing our serving stack against a community vLLM port revealed that our engine decoded 16 sequences *slower* than 12 (328 vs. 354 tokens/s on an 8B model). One configuration change removed the dip (Section III), but the profiler then showed the small-batch kernel consuming 76 % of every decode step while achieving only 2 waves per SIMD-and a nine-variant tiling sweep could not move it. The kernel was at the optimum of its own design, roughly 3× above the floors set by bandwidth and dot-product throughput.

Closing that gap required a new kernel, and the interesting part of the story is not the kernel. Six of seven design variants in our microbenchmark ladder were slower than expected or slower than the baseline, including a textbook software-pipelined double buffer. The reason turned out to be the compiler: HIP's LLVM backend reordered our loads, stores and arithmetic in three distinct ways that each undid the overlap we had written. We could only see this by reading the generated ISA in a compressed form. Once the order was pinned with two inline-assembly idioms, the same source ran 1.57× faster than the production kernel.

Contributions: (1) a reproducible diagnosis procedure for latency-bound GPU kernels-a subtraction ladder plus a compressed ISA order trace-and a catalogue of three compiler reorderings that defeat latency hiding on this toolchain, with their remedies; (2) a quantized small-batch GEMM kernel for gfx906 (Q4_0 and Q8_0 weights, 13–32 columns) with deterministic split-K and 3 waves/SIMD; (3) an evaluation on six models (7B–24B) against the prior kernel, the generic MMQ path, a dense fp16 rocBLAS GEMM, and a vLLM port, with error statistics, perplexity through the new path, and instruction-level accounting of the remaining headroom.

II. Background and Setting

A. Quantized decode matmuls

Weights are stored in blocks of 32 values with a per-block fp16 scale (Q4_0: 4-bit values plus scale, 18 bytes; Q8_0: 8-bit values plus scale, 34 bytes). Activations are quantized on the fly to Q8_1 (8-bit values, fp16 scale and scaled block sum, 36 bytes). A block dot product is an integer sum of four-way byte products followed by one scaling: for Q4_0, $d_w(d_x \cdot \Sigma_i\ q_i x_i - 8 s_x)$, where the second term removes the unsigned nibble offset using the precomputed block sum s_x. Per block of 32 elements and one column this costs 8 `dp4a` instructions plus unpacking and scaling.

B. Hardware

gfx906 has 64 compute units of four 16-lane SIMDs each; a 64-lane wavefront issues one vector instruction every four cycles per SIMD. Latency is hidden only by interleaving wavefronts, and the number of resident wavefronts per SIMD is set by the larger of register and LDS demand: ≤64 VGPRs allow 4 waves, ≤84 allow 3, ≤128 allow 2. There is no native fp32 atomic add. Peak HBM2 bandwidth is 1 TB/s; peak `dp4a` throughput is roughly 55 TOPS. For an 8B-parameter Q4_0 model (4.5 GB of weights) and 16 columns, these floors are about 5 ms (bandwidth) and 5 ms (dot products) per decode step.

Two properties of this card we had to measure because the documentation is wrong or silent about them. First, the L2 is **8 MiB, not the 4 MiB** the vendor tables list: a 6-MiB working set cycled in-kernel sustains 1832 GB/s, more than twice HBM bandwidth, which a 4-MiB cache cannot produce; 16 MiB drops to 927. Second, the latency ladder-56 cycles at vL1D, 141–280 at L2, 629 at HBM (≈370 ns)-sets the scale for everything that follows: one HBM access costs as much as 75 dependent FMAs (8.4 cycles each, measured), so hiding it needs more than 150 independent instructions in flight per CU, which only many waves provide. Streaming bandwidth confirms the same threshold from the other side: 846–850 GB/s with eight or more waves per CU, 569 with four.

C. Software

We work in a fork of llama.cpp with extensive gfx906-specific kernels (the "turbo" fork), compiled with ROCm's hipcc (LLVM 20-based, ROCm 7.x). The production small-batch kernel, "csw" (M10 in the fork's nomenclature), stages weight blocks once per chunk into LDS and lets 4 wavefronts each own a quarter of the columns. Serving uses a Rust server (fork-serve) with a continuous-batching scheduler over the fork's C API. The power limit was fixed at 150 W throughout.

III. Where the Time Went

A. The dip and its trivial part

llama-batched-bench on Qwen3-8B Q4_0 gave 323, 354, **328**, 397 and 464 tokens/s at 8, 12, 16, 24 and 32 sequences. The fork's Q4_0 dispatch routed 13–16 columns to the generic MMQ path because an earlier measurement on a 7B model had found the csw kernel slower there; with the current kernel that was no longer true, and raising the csw limit to 16 columns gave 384 tokens/s (+17 %) on three models. We mention this only because it is the kind of stale default that hides behind every "hardware is the limit" story.

B. The hard part

A kernel trace (rocprofv3) of the 16-sequence decode step attributed 32 of 42 ms to the csw kernel, 2.3 ms to attention and the rest to small operators and launch gaps. The compiler's resource report (`-Rpass-analysis=kernel-resource-usage`) showed 105–117 VGPRs for the 13–16-column instantiations: 2 waves/SIMD. We swept nine tile configurations (rows per block 4/8/16, waves per workgroup 2/4/8/16, chunk sizes) and three `launch_bounds` occupancy targets. Every variant was slower or spilled; the best remained the existing (4 waves, 4 rows, 1 chunk) at 384 tokens/s. Within its design the kernel was optimal, at 3× the floors.

IV. Method: Subtraction Ladder and ISA Trace

We built a standalone HIP microbenchmark with host-side quantization, a double-precision reference on the quantized values, a copy of the csw kernel, and the four matrix shapes of an 8B model (4096×4096, 1024×4096, 12288×4096, 4096×12288; $N = 16$). Each variant is a template instantiation; a run reports median time over 3×200 launches and the maximum relative error.

A. Subtraction variants

Rather than reasoning about which resource binds, we measured it: variants that replace LDS activation reads with register constants, omit weight reads, omit the dot products, skip the compute loop entirely (*staging only*), or skip staging (*VALU only*). The results are not additive, but they rank. For the first new design (Section V, variant V1) on 4096×4096: full 53.6 µs, no activation reads 46.6, no weight reads 55.1, no dot products 47.6, **staging only 34.3**, loop skeleton 8.5. Two thirds of the time was spent moving data into LDS and waiting at barriers-the weights arrived at an effective 275 GB/s-and neither the LDS reads nor the arithmetic were the bottleneck. This redirected the effort from tiling to pipelining.

B. Compressed instruction-order trace

We compile with `--save-temps`, locate the kernel in the AMDGCN assembly, and reduce each instruction to one letter:

`G` global load, `v` wait on loads (`s_waitcnt vmcnt`), `W`/`r` LDS write/read, `d` `v_dot4`, `|` barrier, `^` branch. Runs are collapsed (`d16`). The trace for one loop iteration fits on a line and answers the only question that matters for a pipelined loop: *in what order do loads, waits, compute and stores actually execute?*

```
(a) staging loop with if():   G2 v2 W3 ^ G3 v2 W2 v W |
(b) double buffer, 1st try:   ^2 G4 r28 ^2 v W v W2 v W2
(c) with bound accumulators:  G4 r5 d6 r d2 r d8 r3 d16
```

Fig. 1. Compressed ISA order of the main loop at three stages. (a) The staging loop with a bounds check: loads are issued in two groups with a wait between them-one latency has become two. (b) First double-buffered version: the loads for the next chunk (`G4`) are issued early, but the LDS stores (`W`) that must wait for them were hoisted above the compute (`d128`), which now runs *after* the wait. (c) After binding the accumulators: loads, compute, wait, stores, barrier-the order the source intended.

C. Three reorderings and two remedies

Loads sunk into branches. A per-element guard (`if (b < nblk)`) in the staging loop splits the basic block; LLVM then places each load immediately before its use inside the guarded region. Remedy: make the common case straight-line (the dispatcher guarantees full chunks; the partial tail takes a separate slow path) and place `asm volatile("" ::: "memory")` after the last load so that no memory operation can move across it.

Stores hoisted above compute. In a double buffer, the stores into the *other* LDS buffer depend only on the prefetched registers, not on the current chunk's arithmetic. The scheduler therefore moves them up, directly behind the loads-and with them the `s_waitcnt`. The compute then executes after the wait, exactly as if there were no prefetch (Fig. 1b). This is why our first double buffer was *slower* than the single-buffered kernel: it paid the extra LDS and registers and gained nothing.

Arithmetic migrating across memory barriers. A memory clobber constrains memory operations only; the `dp4a` chain is free to sink below it. The remedy is to bind the values rather than the memory: for each accumulator, `asm volatile("" : "+v"(acc) :: "memory")` placed between compute and stores forces the compute that produces `acc` to complete before the barrier and keeps the stores after it. With this single change the trace became Fig. 1c and the variant became the fastest on all four shapes.

V. The db-GEMM Kernel

Work decomposition. A 256-thread workgroup owns 32 rows and 16 columns. Its four wavefronts split *K*, not the columns: within a chunk of 8 weight blocks, wave *w* processes blocks *w* and *w*+4 for all 16 columns. Each thread accumulates a 2×4 tile of (row, column) products; its four columns are *c*, *c*+4, *c*+8, *c*+12 so that the 128-bit LDS reads of four lanes hit distinct banks. At the end of *K* the four partial tiles are summed through LDS.

Staging. Both weights and activations for the chunk live in LDS, double-buffered. Each thread loads exactly one 18-byte weight block (one 2-byte and four 4-byte loads) and half of one 36-byte activation block, in a straight-line sequence of seven loads followed by one wait. The reduction buffer aliases the staging buffers; total LDS is 20.7 KB. The instantiation uses 83 VGPRs and `__launch_bounds__(256,3)`: 3 waves/SIMD.

Split-K. Narrow matrices (the K/V projection has 1024 rows, i.e. 32 workgroups) cannot fill 64 CUs. A second grid dimension splits *K* into up to 16 ranges so that at least 256 workgroups are resident; partial sums go to a scratch buffer and a second kernel adds them in fixed order. This keeps results deterministic, which gfx906's lack of native fp32 atomics would otherwise forbid.

Variants. 13–15 columns are handled by zero-padding on the 16-wide tile. A 32-wide tile (2×8 per thread, 4-block chunks, waves 0–1 staging weights and 2–3 staging activations, 100 VGPRs) serves 17–32 columns and reads the weights once; an earlier two-group scheme that ran the 16-wide kernel twice was within noise at 24–32 but lost at 20. A Q8_0 kernel (34-byte blocks, no unpacking, 4-block chunks) covers 13–16 columns; above that the two-group scheme loses 11 % on Q8_0 and is disabled.

VI. Evaluation

A. Microbenchmark

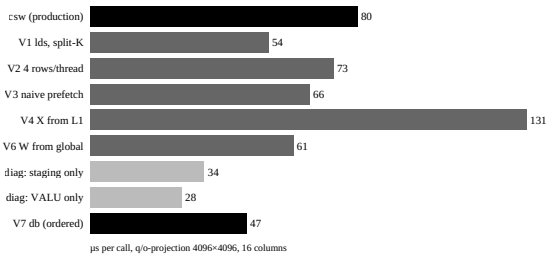

Fig. 2. The variant ladder on the q/o-projection shape (µs per call). Grey: design variants; light grey: subtraction diagnostics; black: production baseline and final kernel. Variants V2–V6 each tested a hypothesis about the binding resource and each was wrong; the diagnostics and the ISA trace located the real one.

TABLE I
Microbenchmark, 16 columns: time per call and error vs. double reference (max|Δ|/max|ref| · NMSE)

Shape (N×K)	csw µs	db Q4 µs	db Q8 µs	hipBLAS fp16	csw err	db Q4 err
4096×4096	80.6	**46.8**	88.0	190	1.1e-7 · 5e-15	1.3e-7 · 8e-15
1024×4096	28.0	**19.9**	29.5	119	1.0e-7 · 5e-15	1.3e-7 · 7e-15
12288×4096	234.2	**131.4**	245.1	1093	1.1e-7 · 6e-15	1.9e-7 · 1e-14
4096×12288	202.4	**140.4**	258.0	604	8.9e-8 · 7e-15	1.9e-7 · 2e-14

hipBLAS: `hipblasGemmEx`, fp16 inputs, fp32 accumulation, on pre-dequantized weights; dequantization time excluded. db Q8 operates on 2× the weight bytes of Q4.

Table I shows the production csw kernel, the two db kernels and a dense fp16 GEMM from rocBLAS on the same shapes. db-gemm Q4_0 is 1.42–1.78× faster than csw per shape (1.57× summed). rocBLAS's skinny-GEMM path is 2.3–8× slower than the quantized kernels even before dequantization, so the dense library is not a viable alternative on this device. Errors are at fp32 rounding level for all kernels; db's NMSE is 1.5–2.5× csw's because of the split-K second pass. Feeding 16 identical activation columns yields bitwise-identical output columns for both kernels.

B. Instruction accounting

Counting the compiled main loop per thread and chunk: db Q4_0 executes 128 `v_dot4`, 52 unpack (and/shift), 56 scaling (cvt/mul/fma) and 15 other vector instructions, plus 28 LDS reads and 23 waits; csw has a nearly identical vector mix (128/48/53). The dot products are 51 % of vector issue slots; for Q8_0 (no unpacking) 65 %. After pipelining, the VALU-only variant of the kernel accounts for 27.5 of 47 µs on 4096×4096: arithmetic is now the largest single component, and roughly half of it is format overhead rather than dot products.

C. End-to-end throughput

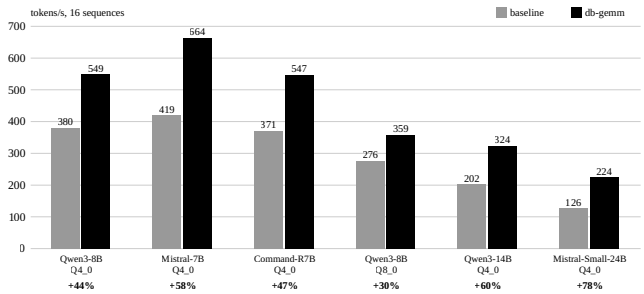

Fig. 3. Decode throughput at 16 concurrent sequences, llama-batched-bench, baseline (same build, `GFX906_DISABLE_DB_GEMM=1`) vs. db-gemm. Six models, 7B–24B parameters.

TABLE II
Tokens/s by number of concurrent sequences (baseline → db-gemm, gain)

Model	B1	B8	B13	B16	B24	B32
Qwen3-8B Q4_0	95→98	321→323	334→443 **+33%**	380→549 **+44%**	397→517 **+30%**	464→555 **+20%**
Mistral-7B Q4_0	111→110	378→380	368→513 **+39%**	419→664 **+58%**	457→583 **+28%**	537→639 **+19%**
Command-R7B Q4_0	95→95	297→297	325→431 **+33%**	371→547 **+47%**	408→498 **+22%**	416→533 **+28%**
Qwen3-8B Q8_0	64→64	241→240	237→304 **+28%**	276→359 **+30%**	296 (off)	346 (off)
Qwen3-14B Q4_0	59→60	178→178	172→210 **+22%**	202→324 **+60%**	214→293 **+37%**	252→370 **+47%**
Mistral-Small-24B Q4_0	40→40	120→119	105→141 **+34%**	126→224 **+78%**	147→201 **+37%**	176→256 **+45%**

llama-batched-bench, flash attention on, f16 KV cache, 64-token prompt (32 for 24B), 128 generated tokens (96/64 for 14B/24B), 40–60 s idle between runs. B1 and B8 do not use the new kernel.

Table II and Fig. 3 cover six models. Gains at 16 sequences range from 30 % (Q8_0) to 78 % (24B) and *grow with model size*: larger hidden dimensions give longer K ranges per workgroup, less split-K overhead, and relatively more expensive MMQ baselines. On Qwen3-8B the 16-sequence step fell from 42.6 to 29.0 ms and the kernel's share from ~32 to ~18 ms. At 13 columns the zero-padded tile still wins by 22–39 %. The 32-wide tile lifts 24 and 32 sequences by 19–47 %; on the 8B model 16 sequences now decode faster than 24, because 17–32 columns still pay for a 100-VGPR tile at 2 waves/SIMD.

D. Through the serving stack

Measured over HTTP through fork-serve's batch scheduler (32 slots, q8_0 KV, 512 generated tokens per stream, 16 concurrent streams, second of two runs), Qwen3-8B Q4_0 went from 332 to 451 tokens/s (+36 %); at 13 streams from 299 to 357. A community vLLM port for gfx906 (AWQ weights, fp16, Triton attention) measured 329 tokens/s at 16 and 383 at 32 streams on the same hardware with the same client; the serving stack is now ahead at 16 (451) and roughly level at 32 (395 vs. 383, with 32 streams limited by the scheduler rather than the kernel). Before this work the same comparison had our stack behind vLLM at 16 streams (317) and, owing to a scheduler bug found in the same campaign-non-contiguous sequence IDs fragmenting each step into sub-batches- erratically between 125 and 153 at 32.

E. Correctness in the model

Twelve new large MUL_MAT cases in llama.cpp's test-backend-ops (1024–12288 rows; 13, 15, 16, 24, 32 columns; K 4096/12288; Q4_0 and Q8_0) pass against the CPU reference, with a debug trace confirming dispatch through the new kernel; the full MUL_MAT suite reports no failure. Perplexity on wikitext-2 (8 chunks of 512, prompt processed in 16-token micro-batches so that every prefill matmul runs through db-gemm; 7 888 dispatches per run) is 10.518 ± 0.730 for the baseline and 10.528 ± 0.733 with db-gemm on Q4_0, and 10.215 vs. 10.256 (±0.71) on Q8_0: differences of 0.1–0.4 %, an order of magnitude inside the estimate's uncertainty and consistent with a changed summation order. Sixteen identical greedy sequences produce two distinct continuations with db-gemm where the baseline produces one; the divergence occurs at a near-tie 114 characters in, and the kernel itself is column-symmetric, so we attribute it to rounding-order sensitivity at ties rather than to a defect.

F. Extensions along the batch axis

With the 16-column kernel as template we covered the neighbouring regimes (Table III). An 8-column tile (2×2 per thread)

wins 7 % at 8 sequences and is neutral at 5–6; the zero-padded 16-column tile already beats the previous kernel from 9 columns on, including the previous kernel's worst point (11 columns: 250 → 382). A Q8_0 32-column tile lifts 24 sequences by 36 %. Relaxing the chunk granularity from 256 to 128 elements (a 4-block tail with clamped weight index and zeroed activations) admits hidden sizes such as 2688 and 3712, which occur in the Nemotron-H family.

TABLE III
Extensions (tokens/s, baseline → new; Qwen3-8B unless noted)

Change	Regime	Result
8-column tile (M15)	B8 / B5–6	322 → 345 (+7%) / neutral
padded 16-tile from 9 cols (M15)	B9 / B10 / B11 / B12	302 → 320 / 336 → 352 / 250 → 382 / 353 → 412
Q8_0 32-column tile (M12d)	Q8_0 B24 / B32	302 → 412 (+36%) / 345 → 376 (+9%)
K % 128 tail chunk (M16)	Nemotron-H 30B-A3B B16 (dense part)	198 → 205 (+4%)
MMQ Q4_0 register-first staging (M13a)	prefill pp512 / PP2048	978 → 1114 (+14%) / 931 → 1053
MMQ Q4_0 LDS stride 36, b128 reads (M13c)	prefill pp512	1114 → 1123 (+0.8%)
MMQ register double buffer X+Y (M13b)	prefill	**spills at 128 VGPR: −4% (off)**
Q4_K tile (M14)	phi-4 Q4_K_M B13 / B16	203 → 214 (+5%) / 247 → 258 (+4%)
MoE expert-gather tile (M17)	Nemotron-H B8 / B16	**171 vs 175 / 205 vs 206 (neutral, opt-in)**

G. Attention at long context: the occupancy trap

All results so far use short contexts, where attention is under 10 % of the step. At 32k tokens it dominates: a decode-depth profile attributes 55 % of the step to the prefill flash-attention kernel. The production server stores its KV cache in Q8_0, which selects a gfx906-specific tile kernel-but that kernel was tuned for decode (one query column) and ran at *half* the throughput of the generic f16 tile kernel on the many-column prefill (246 vs. 485 tokens/s at 32k). The cause was the same class of mistake as the register ceiling in Section III, inverted: the head-dim-128 configurations for 32 and 64 query columns were compiled at an occupancy target of 2 waves/SIMD, which capped registers and throttled a kernel that wanted more of them. Lowering the target to 1 wave lifted prefill by 40–48 % (16k: 397 → 556, 32k: 246 → 363 tokens/s) with decode unchanged-decode uses a different column tile-and all 3344 flash-attention reference tests passing. The lesson mirrors the decode kernel: occupancy is a two-sided knob, and forcing it up throttles a latency-bound kernel exactly as forcing it down would starve a bandwidth-bound one; the only way to know which applies is to measure both directions.

The decode side of the same kernel then produced a third variant of the lesson. Profiling showed that q8-KV decode does not use the 16-column tile we had assumed but a four-column one (GQA packing 1×4). Sweeping its configuration, neither occupancy target nor K-batch size mattered-the winning change was the *workgroup width*: 128 threads (two wave64 per group) left the CU idle on a single-column decode, and 256 threads raised 32k-context decode from 55.4 to 61.3 tokens/s (+10.6 %) with decode-path tests unchanged. Occupancy, register ceiling, workgroup width: three knobs that all present as "too few waves in flight", each decisive for a different kernel, none deducible from the source.

H. Prefill: the same lesson in the tiled GEMM

A kernel trace of a 2048-token prompt attributed 83 % of GPU time to the tiled integer GEMM (MMQ) and 8.5 % to attention. Its Q4_0 tile loader interleaved each global load with its LDS store, the pattern of Fig. 1a; the fork had already fixed this for Q8_0 tiles. Loading the whole tile into registers first, then storing (M13a), gave +14 % on pp512. Going further is harder than in the decode kernel: the loop is issue-bound at 2 waves/SIMD (100–128 VGPRs), its hot block holds 256 `v_dot4` against 112 scaling and 24 unpacking instructions, and a register double buffer of the next tile spills at exactly this register ceiling (M13b, −4 %). Vectorising the LDS reads by widening the row stride from 33 to 36 ints (M13c) is worth under 1 %. The register-first staging transfers to the K-quant tile loaders as well (+7.7 % on phi-4 Q4_K_M).

I. The dispatch is part of the kernel

Comparing against a current upstream build (b10524, built for the same target) put our work in perspective and produced one concrete import. On the everyday models the fork leads by 20–78 % in throughput and by 4× in long-context decode-at 32k tokens upstream loses 37 % of its decode rate (78.9 → 49.4 tokens/s) where the fork loses 5 % (95.7 → 91.0), the accumulated effect of the attention work above. Upstream won exactly one measurement: the 32k prefill with q8-quantized KV (489 vs. 376 tokens/s).

The cause was not a better kernel but a different dispatch decision. Upstream converts the quantized KV cache to fp16 once per attention call and runs the fp16 tile kernel; we ran a q8-specialised tile kernel for every shape. Their choice is right for prefill (many query columns amortise one conversion) and catastrophic for decode (14.9 tokens/s: the whole cache converted per token), ours is the mirror image. Routing by column count-specialised kernel up to 32 columns, conversion path above-takes both: prefill at 4k/16k/32k rose from 903/536/376 to 1050/726/514 tokens/s while decode stayed at 101/60. The lesson generalises beyond this case: a kernel that is optimal in isolation can be the wrong choice at a shape the author never profiled, and the routing table deserves the same scrutiny as the inner loop.

J. Six negative results

K-quants. A Q4_K port of the kernel (scales and mins decoded once per row during staging) is only 15 % faster than the MMQ Q4_K path at kernel level and 4–5 % end to end on phi-4 Q4_K_M. The reason is the baseline, not the port: llama.cpp stages K-quant tiles unpacked to 8 bits, so the tiled path pays no unpacking in its inner loop and sits much closer to its floor than the Q4_0 path did. We did not port Q5_K/Q6_K for decode; for prefill, the same register-first tile-staging that helped Q4_0 (Section VI.H) also helped the Q4_K and Q6_K MMQ tile loaders, +7.7 % on phi-4 Q4_K_M pp512.

Mixture of experts. For batched decoding of an MoE model the per-expert matmuls have 0–3 real columns each; we implemented an expert-gather variant (one expert per grid slice, token columns gathered through the index arrays that the MMQ path already builds, outputs scattered). It is correct (72 MUL_MAT_ID tests) but neutral on Nemotron-H 30B-A3B, as is raising the batch limit of the existing matrix–vector MoE path from 4 to 8 tokens: all three paths land within 2 % of each other because, on a 16 GB card, six of this model's expert layers run on the CPU and their cost scales with the batch. The GPU kernel is not the bound.

The server is not the bottleneck. We suspected the serving stack of losing 20 % against the engine at 16 sequences, and budgeted Rust work for it. Measured at matched generation length, the scheduler adds 0.1–0.7 ms per step across 8–32 sequences-under 1 %; the apparent gap was an artefact of comparing a 512-token server run against a 128-token engine run. HIP graphs, our planned remedy, cannot help a 0 ms overhead.

Memory overclocking is capped three times over. With the vendor tool unlocked we raised the HBM clock through the PowerPlay table. Three limits appeared in sequence: the SMU refuses any state above ~1090 MHz and silently falls back to 800 under load (a firmware/training limit-raising every table ceiling changes nothing); read bandwidth saturates at ~890 GB/s after +4 % (the fabric, not the DRAM-only writes scale, +13 %); and re-uploading the table costs 1.5–2.5 % by itself, which we caught only by re-uploading the *unmodified* table as a control. Net effect on inference: zero to −4 %. ECC counters stayed clean throughout, so the memory could take more-the controller will not.

Four-bit activations remain out of reach, but not by much. The v_dot8 instruction runs at full rate yet reads both operands as 4-bit nibbles, which is why it went unused (Section VI.J, first item). Reformulating the kernel with a zero-point and a per-group weight-sum side table makes it usable: measured 1.76× over the Q4_0×Q8_1 core (905 vs. 513 GBlockDot/s, 19 vs. 24 VALU instructions per block). We then measured the quality cost before writing the kernel, by rounding activations to int4 inside the existing quantizer-five schemes in an afternoon. The best (asymmetric, zero-point, group 16, with the output/down/attention-output projections left at 8 bit) costs +0.81 % perplexity over 150 chunks: just outside our ±0.7 % band. Naive schemes cost 3–6 %, and asymmetric quantisation *without* a zero-point is worse than symmetric because post-SiLU activations are full of exact zeros that then miss the grid. The path into the band is a SmoothQuant-style offline fold of channel scales into the weights; we did not build it.

Placement. During the Q4_K work the 13-sequence throughput was bimodal across process starts (214 vs. 168 tokens/s, 3:1), with the baseline stable. The split-K scratch buffer was the variable: aligning it to 2 MiB removed the effect (5/5 runs at 214). We had not seen this with Q4_0 and would have misattributed it to thermal noise without the repeat protocol.

VII. When the Kernel Is Right and the Policy Is Wrong

Kernel work has a natural end. Ours arrived when the remaining levers turned out to be *policies*-decisions the server makes per request-rather than instructions. The clearest case is speculative decoding, which this model family supports through a draft head.

Its value is the product of two quantities, both measurable at runtime: the acceptance rate of the drafts, which follows the character of the text (90–100 % on code and repetition, 49–79 % on free prose), and the marginal cost of a verification row, which follows the architecture. On a dense resident model the weights are in VRAM anyway and one extra row is nearly free: speculation gains 15–40 % across every prompt type. On a mixture-of-experts model with experts offloaded to the host, every row triggers transfers over a 14 GB/s link, and a rejected draft costs real time: the same speculation *loses* 23 % on prose. The break-even sits near 65–70 % acceptance for the dense models and 85 % for the offloaded one.

Because the product flips within a single session, we moved the decision into the server. A sliding window over the last 16 verified drafts disables speculation when acceptance drops below the threshold, with an exponentially growing lock-out so that continuous prose is rarely re-probed while a switch to code is still detected. The measurable part is not the window but what is suspended: suppressing only the verification row recovered a third of the loss (66.9 → 72.1 tokens/s); additionally *deferring the draft-head pass*-buffering its input pairs and replaying them in one batched decode when the gate reopens-recovered nearly all of it (84.3, against 87.1 with speculation off entirely).

The batched path needed a different criterion, and finding out why was instructive. There the head runs anyway for the other streams, so a draft costs only its verification row; the acceptance threshold calibrated for the solo case suspends streams whose draft would have been almost free, and cost 23 % on a good prompt. Replacing the threshold with a direct measurement-two moving averages of nanoseconds per emitted token, with and without drafts, and a periodic counter-probe-removes the constant altogether and lifts multi-stream prose from 110 to 142 tokens/s.

Two methodological traps are worth recording, because both produced convincing wrong numbers. The cost window must cover *all* the work: our first version timed only the head phase while the draft rows are paid for in the main decode, and the regulator duly chose wrong. And deferred work must not be charged to the wrong bucket: buffering the head pass while the gate was closed made the eventual batched decode land in

the "with draft" average, poisoning exactly the comparison the gate relies on. Finally, evaluating an *adaptive* component with a fresh process per arm measures its cold start, not its behaviour-that mistake made a 29 % gain read as 1.7 % until we ran six consecutive measurements in one process and watched the acceptance rate fall to zero as the regulator settled.

VIII. Discussion and Limitations

What generalises. The diagnosis procedure-measure by subtraction, then read the order the compiler actually emitted-is toolchain-agnostic, and the three reorderings are consequences of standard LLVM scheduling rather than of anything gfx906-specific. Software-pipelined HIP kernels that show no benefit over their unpipelined form should be checked with a trace like Fig. 1 before any other hypothesis is entertained. The accumulator-binding idiom is cheap, portable across AMDGPU targets, and does not inhibit the scheduler elsewhere in the loop.

What does not. All measurements come from one device, one driver stack and one compiler version. We have not verified that the reorderings reproduce on gfx908/gfx90a (whose MFMA units would in any case change the kernel design) or under future ROCm releases. The kernel design assumes a 64-lane wavefront and a 64 KB LDS and is tuned for dimensions that are multiples of 256 in K and 32 in N; other shapes fall back to the previous path. Q5_K and Q6_K are not covered; Q4_K and Q8_0 beyond 16 columns are (Section VI.F–H), with smaller gains.

Baselines. Our baseline is the fork's own previous kernel and its MMQ path with their default tiles; we did not retune MMQ tiles for N = 16, although forcing MMQ or the rocBLAS path end-to-end at 16 columns gave 318 and 327 tokens/s respectively against 384 for the old small-batch kernel, so the baseline is the strongest of the three. The vLLM comparison uses a third-party port with a different weight format (AWQ) and is reported as context rather than as a controlled comparison.

Remaining headroom. The kernel sits at 47 µs on 4096×4096 against ~10 µs of combined floors. Half of its vector instructions are nibble unpacking and per-block scaling; removing them requires either a weight layout with pre-separated nibbles (at 2× the memory traffic) or integer tricks we have not found on this ISA. A second axis is the 32-wide tile's register pressure. We also note that thermal state is a first-order confounder on a power-capped card: the same configuration measured 425 and 547 tokens/s depending on whether it ran at the end of a ten-minute series or after a one-minute pause. All reported numbers use pauses; we recommend reporting junction temperature alongside throughput.

IX. Related Work

llama.cpp's CUDA/HIP backend provides the three-regime structure (MMVQ, small-batch, MMQ) we build on; its MMQ tiles are designed for N ≥ 32 with dp4a on architectures without tensor cores. vLLM's continuous batching and paged attention define the multi-sequence serving regime; the gfx906 community ports replace its Triton GEMMs with GPTQ/AWQ-specific kernels. Software pipelining and double buffering for GEMM are textbook (CUTLASS, rocBLAS/Tensile), but those implementations control instruction order with explicit assembly or intrinsics; the failure mode we document -a C++ source whose intended overlap is rewritten away-is the cost of writing such kernels at source level, and inline-assembly barriers to pin schedules have long circulated as folklore in CUDA and HIP communities without, to our knowledge, a published account of the specific reorderings and their detection.

X. Conclusion

A matrix-core-less GPU from 2018 can decode 16 concurrent sequences of an 8B model at 549 tokens/s and a 24B model at 224, 44–78 % above the previous state of the same software and above a vLLM port on the same card. The kernel that delivers this is not exotic: LDS staging, a double buffer, split-K. What made it work was seeing that the compiler had quietly unpipelined four earlier attempts, and a two-line remedy that pins the order. The same staging discipline transferred to the prompt-processing GEMM for +15 %. We offer the subtraction ladder and the compressed ISA trace as a routine check for any latency-bound kernel whose prefetch "does not help," and the three negative results as a reminder that the method finds bounds as reliably as it finds gains.

Two further observations survive this hardware. The first is that dispatch decisions deserve the same scrutiny as inner loops: the only measurement our specialised attention kernel lost against a generic upstream build was one where upstream had made a better *routing* choice, and adopting it cost twenty lines. The second is that kernel work terminates. Once the instruction schedules were tight, the remaining levers were policies evaluated per request-whether to speculate, at what batch width-and those belong under a regulator that measures its own cost, not under a constant chosen on a good day. The card is old, undocumented in places, and out of vendor support; it also still has room, and most of what we found was visible only because we measured in both directions and kept the results that contradicted us.

Artifacts

Kernel: `ggml/src/ggml-cuda/gfx906/matmul/mmvq-db-gemm.cuh` in llama-cpp-gfx906-turbo; dispatch in `mmvq.cu` and `ggml-cuda.cu`; tests in `tests/test-backend-ops.cpp`. Microbenchmarks and ISA-trace script: `scratchpad/m12-lds-gemm/bench.hip`, `bench2.hip`. Raw measurements, including the discarded variants and the thermal artefacts: BENCHMARKS.md, campaigns AX–BC. Kill switch: `GFX906_DISABLE_DB_GEMM=1`; dispatch trace: `GFX906_MINI_GEMM_DEBUG=1`.

References

[1] G. Gerganov et al., "llama.cpp: LLM inference in C/C++," github.com/ggml-org/llama.cpp, 2023–2026.

[2] W. Kwon et al., "Efficient memory management for large language model serving with PagedAttention," in *Proc. SOSP*, 2023.

[3] nlzy, ttdxq, ai-infos, "vLLM for AMD gfx906 GPUs," github.com/ai-infos/vllm-gfx906-mobydick, 2025–2026.

[4] AMD, "'Vega' 7nm Instruction Set Architecture Reference Guide," 2019.

[5] T. Dettmers et al., "LLM.int8(): 8-bit matrix multiplication for transformers at scale," in *Proc. NeurIPS*, 2022.

[6] J. Lin et al., "AWQ: Activation-aware weight quantization for LLM compression and acceleration," in *Proc. MLSys*, 2024.

[7] NVIDIA, "CUTLASS: CUDA templates for linear algebra subroutines," github.com/NVIDIA/cutlass.

[8] S. Haas, "Chained acceptance is a model property," preprint, 2026 (companion work on the same hardware).