

Hardware–Software Co-Design for Grouped-Query Attention on AHBM

Mukesh Garg

Jiyeon Lee

Yangwook Kang

AGI Computing Lab, System Technology Group

H1 2026

Executive Summary

Grouped-Query Attention (GQA) has largely replaced GPT-3-style multi-head attention in modern decoder-only LLMs (e.g., Llama 3 and Mistral) due to its reduced memory and bandwidth requirements. Mapping GQA efficiently onto AHBM introduces three architectural requirements: optimized placement of KV caches and weights to minimize inter-PE communication, low-overhead support for unavoidable cross-PE traffic, and efficient pipelining of memory accesses and computation within each PE.

To address these requirements, this report introduces three hardware–software co-design mechanisms: GQA-aware placement of KV caches and weights across TCM, SRAM, and HBM; PE_IPCQ, an efficient on-device collective communication primitive; and a composite-command GEMM pipeline that tightly pipelines memory and compute operations within each PE under PE_SCHEDULER control. Kernbench-based evaluation show up to 38 % lower collective communication overhead, 25 % to 35 % lower all-reduce latency than a mesh-based interconnect, and up to 78 % of peak MAC efficiency for compute-intensive GEMM kernels. These results demonstrate that the fused GQA kernel can efficiently exploit AHBM’s HBM bandwidth.

While GQA serves as the motivating workload in this study, the resulting architectural mechanisms are broadly applicable across the AHBM software stack. PE_IPCQ provides a scalable communication substrate for collective operations and communication-intensive workloads, while composite-command execution enables efficient fusion of memory movement, GEMM kernels, normalization, and other element-wise operations. Together with the hierarchical data-placement framework, these capabilities form a reusable foundation for future AI kernels and communication libraries on AHBM.

1 Introduction

The performance of a large language model on an accelerator is decided less by peak FLOPS than by how well the kernel exploits the memory system and the interconnect. This is especially true of attention, which in the decode

phase reads a large KV cache to do a small amount of arithmetic. Optimizing such kernels is fundamentally a *codesign* problem: the algorithm (how to tile, fuse, and reduce) and the hardware (what issue mechanism, what collective engine, what memory hierarchy) have to be designed against each other. Optimizing only the software leaves the hardware idle; adding hardware that the software cannot reach is wasted area.

The 1H focus: attention. This half’s work targets attention. Multi-head attention (MHA) was studied previously and is taken here as the established baseline. The new work is FlashAttention-style tiling and Grouped-Query Attention (GQA)—the form used by modern long-context decoders, in which several query heads share a single KV head to shrink the KV cache. Realizing a fast *fused* GQA kernel, however, is not a single optimization. It decomposes into two enabling optimizations that we studied separately and then brought together inside the fused kernel:

1. **GEMM optimization** via a composite command (§3): a way to issue a tiled matrix multiply as one self-routing pipeline so the MAC array stays fed without per-tile command overhead. Attention’s $Q \cdot K^\top$ and $P \cdot V$ products are exactly such GEMMs.
2. **Communication optimization** via PE_IPCQ (§4): a per-PE on-device collective engine that performs all-reduce overlapped with compute. Attention’s multi-user and sequence-parallel KV reductions are exactly such collectives.

The report therefore reads as *two enablers building to one capstone*: GEMM (§3) and all-reduce (§4) are developed and measured on their own, and then fused GQA (§5) shows them operating together. Before the results, §2 describes the KernBench platform—why a software-stack-independent, source-level simulator is the right tool for this question, how it executes a kernel, how it computes latency and how accurate that is, and the exact hardware configuration used throughout. We close with a cross-cutting discussion of which hardware changes are worth their cost (§6), a conclusion (§7), and the 2H agenda (§8): extending from attention to the feed-forward and mixture-of-experts layers, and to the compute/data distribution

questions that full LLM decoding and agentic workloads raise.

2 The KernBench Platform

All results in this report are produced on *KernBench*, a system-level, discrete-event simulator for LLM kernels running on SIP-based AI accelerators. This section explains why the platform exists, how it executes a kernel, how it computes latency and how accurate that number is, and the concrete hardware configuration used for every experiment that follows.

2.1 Why KernBench: source-level kernels without a software stack

In a production end-to-end (E2E) stack, kernel performance is entangled with every layer above the hardware: the compiler’s tiling and scheduling choices, the framework’s operator dispatch, the collective library, and the runtime. Good E2E numbers require *all* of those layers to be co-optimized, which makes it hard to answer a narrower but more fundamental question: *given the hardware, how fast can a well-written kernel be, and which hardware features actually make it faster?*

KernBench is built to answer exactly that question. Kernels are written and executed at the *source level*—as algorithmic descriptions in a small tile-oriented kernel API—with no dependency on a compiler or any other software-stack layer. The simulator takes the kernel and a hardware topology and reports the latency that the modeled hardware would deliver. This isolation is deliberate: it lets us study algorithm-level optimizations (how to tile a GEMM, how to schedule a collective, how to fuse an attention kernel) and the hardware features that support them, without the confound of compiler maturity or framework overhead. The cost is that KernBench numbers are *not* E2E latencies; they are the achievable-kernel latencies that an ideal software stack would expose.

2.2 Execution model

KernBench is layered along the flow of a request:

- The **runtime API** is host-facing and topology-agnostic—it deploys tensors and launches kernels but knows nothing about routing or interconnect.
- The **simulation engine** schedules discrete events, routes every request through the modeled graph, and tracks completion via correlation IDs.
- The **components** are device-side nodes that model hardware behavior: the per-PE blocks (scheduler, DMA, GEMM and vector-math engines, TCM, IPCQ), the NoC routers, the HBM controllers, and the inter-chiplet links.

The topology is compiled once at configuration time into an authoritative graph of components and links; it is never mutated during a run. Within a PE, work is expressed as *composite commands*: a single command carries an ordered pipeline of operations (DMA_READ → FETCH → GEMM/MATH → STORE → DMA_WRITE) that the PE scheduler tiles and streams. This composite mechanism is the substrate for the GEMM optimization (§3) and, combined with on-PE collectives, for fused attention (§5). Data and timing are handled in two passes, so that a kernel’s numeric results and its latency are computed consistently but independently.

2.3 Latency model and its accuracy

KernBench obeys a set of golden invariants that keep its latency numbers physically meaningful. End-to-end latency is computed *strictly by explicit traversal* over modeled components and links: every routed request incurs latency greater than zero, routing is deterministic, and every valid request flow has explicit connectivity. There are no hidden shortcuts, implicit waits, or magic delays—if a delay exists in the result, it came from a scheduled event on a modeled component or link.

Latency accumulates from three kinds of contributions: per-node fixed overheads (each component carries an **overhead_ns**), per-link transfer time, and per-service occupancy. The interconnect is modeled at fine granularity. Each directed link serializes traffic through a bandwidth-limited FIFO, so a busy link delays later flits. Payloads are decomposed into fixed-size flits (default 256 B bursts) that arrive at $\text{prop} + \text{flit_bytes}/\text{bw}$ intervals, so link bandwidth throttles arrival rate rather than being applied as a lump sum. HBM is modeled with per-pseudo-channel parallelism: a stateless array of channel-availability timestamps with address-based channel selection captures bank-level concurrency.

The cost of *issuing* a command is modeled structurally rather than with a per-operation calibration table. The PE control processor charges, per command,

$$d_{\text{cmd}} = \text{FIXED} + b_{\text{logical}} \cdot R,$$

where b_{logical} is the command’s hardware-logical byte size, FIXED captures the fixed per-command cost (queue-tail update, completion registration) and R captures the per-byte cost of serializing the command descriptor into the scheduler queue. The default anchoring (FIXED = 40 cycles, $R = 0.0625$ cycles/byte, i.e. 16 B cycle^{−1}, at 1 GHz) places a typical composite at roughly 43 ns, and a hard cap on a composite’s descriptor size prevents the model from rewarding arbitrarily large fused commands beyond what real descriptor queues accept. The dispatch cost is enabled and tuned per topology; in the configurations measured here, command issue is not the bottleneck—data movement is—so this term stays small relative to DMA and collective time.

Table 1: Modeled hardware configuration (shared by all experiments).

Parameter	Value
<i>Hierarchy</i>	
SIPs	2 (1D ring)
CUBEs per SIP	16 (4×4 mesh)
PEs per CUBE	8 (4 corners $\times$ 2)
PEs total	256
<i>Processing element (PE)</i>	
GEMM engine peak	8 TFLOP s ⁻¹ (f16)
TCM (on-PE)	16 MB, 512 GB s ⁻¹ R/W
kernel scratch	1 MB
DMA engines	1 read + 1 write
CPU / scheduler overhead	2 ns / 1 ns
<i>Memory (per CUBE)</i>	
HBM capacity	48 GB (8 slices)
HBM aggregate BW	1024 GB s ⁻¹
HBM pseudo-channels	64 (8 per PE), 32 GB s ⁻¹ each
SRAM (shared)	32 MB, 128 GB s ⁻¹ link
HBM burst	256 B
<i>Interconnect</i>	
Intra-CUBE NoC link	256 GB s ⁻¹ , 2 ns/router
Inter-CUBE (UCIe PHY)	512 GB s ⁻¹ , 8 ns, XY routing
Inter-SIP (PCIe)	768 GB s ⁻¹ per endpoint
<i>Command-issue cost model (defaults)</i>	
FIXED per command	40 cycles
per-byte rate R	0.0625 cycles/byte (16 B cycle ⁻¹)
composite size cap	1024 B

How accurate is all this? The model is precise about the things that dominate kernel latency on this class of hardware: link bandwidth occupancy and serialization, HBM channel parallelism, flit-level streaming, and per-component switching overhead. The GEMM study (§3) provides a direct check: the measured MAC efficiency tracks the analytic (theoretical) efficiency within roughly 10% to 20% across a wide range of tile counts, with the gap attributable to pipeline fill and DMA effects that the analytic model omits. The known simplifications—idealized arbitration, no thermal or refresh effects, fixed burst granularity—are the price of a deterministic, inspectable model; they bound the absolute accuracy but do not distort the *relative* comparisons (tiling A vs. B, topology X vs. Y) that this report is built on.

2.4 Modeled hardware configuration

Table 1 summarizes the hardware configuration used for every experiment in this report. It is read directly from the simulator’s topology description; per-experiment workload parameters (matrix shapes, collective sizes, sequence lengths) are stated in their respective sections rather than here.

3 GEMM Acceleration via the Composite Command

3.1 Why it is needed

GEMM is the compute core of every transformer block—the QKV projections, the attention score and context products, and the feed-forward matrices are all matrix multiplications. On a tiled accelerator a single logical GEMM expands into many hardware tiles, and each tile must be read from HBM, fetched into the register file, computed, stored, and written back. If every one of those tile-stage steps were an independently issued command, two costs would dominate. First, the host and the PE control processor would pay a per-command issue overhead $O(\text{tiles} \times \text{stages})$ times, which for a deep- K reduction is hundreds to thousands of issues. Second, with stages dispatched one-at-a-time the scheduler cannot overlap the DMA of the next tile with the compute of the current one—the pipeline never fills, and the MAC array sits idle waiting for data. The hardware question is therefore: what issue mechanism lets a single GEMM saturate the MAC array without drowning in command overhead?

3.2 Design

The answer is the *composite command*. A single command carries the ordered tile pipeline

DMA_READ $\rightarrow$ FETCH $\rightarrow$ GEMM $\rightarrow$ STORE $\rightarrow$ DMA_WRITE,

and the PE scheduler splits the payload into hardware tiles (here $32 \times 64 \times 32$), emitting one tile token per tile. Subsequent stages are reached by *token self-routing* between the on-PE engines, so a tile flows DMA $\rightarrow$ fetch $\rightarrow$ GEMM $\rightarrow$ store without returning to the scheduler between stages. Because the whole pipeline is described by one command, the issue cost is paid once per GEMM rather than once per tile-stage, and the scheduler is free to keep every stage busy on different tiles simultaneously—tile i ’s GEMM overlaps tile $i+1$ ’s DMA read. A multi-operation composite additionally lets an epilogue (for example a vector-math step) ride the same tile loop, firing per K -tile, per output tile, or once per kernel according to its declared scope; this is what later allows an attention kernel to fuse its softmax work into the GEMM pipeline (§5).

3.3 Results

We sweep eight GEMM shapes spanning square, tall, wide, and deep- K geometries, under three operand-staging variants (ref_ref, both operands streamed from HBM; load_ref, one operand resident in TCM; load_load, both resident). Figure 1 reports MAC utilization and efficiency, and Figure 2 breaks the kernel into per-stage engine busy time.

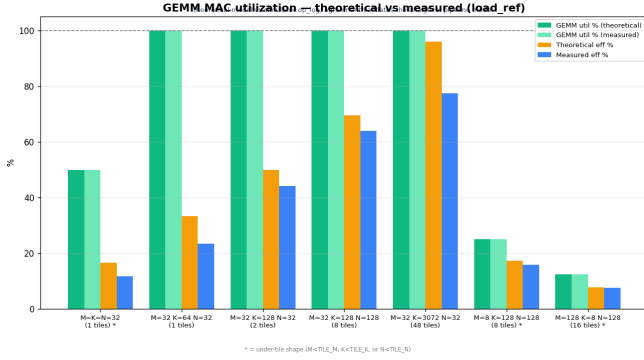


Figure 1: GEMM MAC utilization and efficiency, theoretical vs. measured. Tile-fill sets the ceiling: under-tile shapes (marked *) such as $M=K=N=32$, $M=8$, and $K=8$ cannot fill the MAC tile and cap at 50 %, 25 %, 12.5 %. For tile-filling shapes, efficiency climbs with tile count—from ~23 % at one tile to ~78 % measured at 48 tiles ($K=3072$)—and the measured bars track the analytic prediction within 10 % to 20 %.

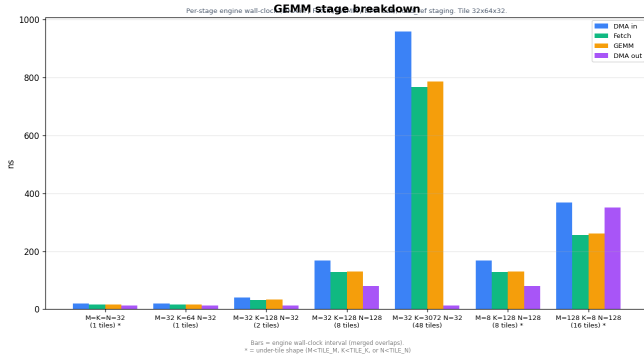


Figure 2: Per-stage engine wall-clock (DMA in, Fetch, GEMM, DMA out) under `load_ref` staging. For the deep- K shape ($K=3072$, 48 tiles) the Fetch and GEMM stages are large and comparable (~770 and 785 ns) while DMA-out is negligible—a compute-rich, well-pipelined regime. For the low-reuse shape ($M=128$, $K=8$, $N=128$) DMA-out grows to ~350 ns and compute is small—a data-movement-bound regime.

Two regularities stand out. (i) Utilization is governed by how completely the problem fills the MAC tile: the three under-tile shapes are hard-capped well below 100 %, independent of how the kernel is issued. (ii) Among tile-filling shapes, efficiency is governed by tile count—more tiles amortize the one-time pipeline fill and issue cost, so the deep- K shape reaches ~78 % of peak while a single-tile shape reaches only ~23 %. The stage breakdown explains why: with 48 tiles the GEMM and Fetch stages overlap and stay busy, whereas the low-reuse shape spends most of its time moving data in and out.

3.4 Analysis and meaning

The composite command does not manufacture bandwidth or MAC throughput; it removes the two software-

shaped obstacles between a GEMM and its hardware roofline. By paying issue cost once per GEMM and chaining tile-stages through token self-routing, it lets the scheduler keep the pipeline full, so compute-rich shapes actually reach the efficiency their arithmetic intensity allows (~78 % measured at 48 tiles), and data-bound shapes actually reach their DMA bound instead of stalling on command overhead. The close agreement between measured and theoretical efficiency (Figure 1) is also the report’s primary validation that KernBench’s latency model is faithful in the regime that matters. The hardware implication is concrete: a single-command, self-routing tile pipeline is the issue mechanism that makes the MAC array usable, and it is a prerequisite—not a luxury—for the fused attention kernel of §5.

4 All-Reduce Acceleration via PE_IPCQ

4.1 Why it is needed

Distributing a transformer across devices turns every tensor-parallel layer into a collective: partial results computed on different PEs, CUBEs, and SIPs must be summed and redistributed with an all-reduce. If that collective is handled by the host or by a generic DMA path, three problems appear. The reduction traffic competes with the kernel’s own compute DMA on the same links, causing head-of-line blocking; there is no efficient peer-to-peer ring primitive, so data takes extra hops; and the ordering is hard to make deterministic. The hardware question is how to perform collectives *on the device*, overlapped with compute and reproducible run-to-run.

4.2 Design

KernBench models a dedicated per-PE collective engine, **PE_IPCQ** (inter-PE communication queue). It is a control-plane block: it owns the ring-buffer address arithmetic, head/tail pointers, peer-pointer caches, backpressure, and the four-direction (N/S/E/W) neighbor map, with eight ring buffers per PE (four directions $\times$ {tx, rx}). Crucially, PE_IPCQ does *not* move data itself—it delegates the actual transfer to PE_DMA, keeping a clean control/data split. To stop collective traffic from blocking compute, PE_DMA is extended into a two-channel virtual-channel model: `vc_compute` carries tile load/store for GEMM and vector math, `vc_comm` carries IPCQ sends, each with an independent state machine. The same physical link is shared but progresses in chunks (256 B), so a large GEMM DMA does not lock the link end-to-end against a pending reduction. On top of this substrate the collective runs a hierarchical local-reduce / global all-reduce-broadcast schedule across whatever inter-device topology the configuration specifies.

Table 2: All-reduce latency (ns) across six devices, by topology and per-PE payload. Lower is better; the torus wins at every size.

Bytes/PE	2D mesh	Ring 1D	2D torus
256	2667	2365	1701
4,096	4450	4082	3038
16,384	8900	8217	6403
65,536	26705	24766	19865
98,304	38574	35798	28840

Multidevice allreduce (ring, Mesh, 2DTorus) vs FSIM latency

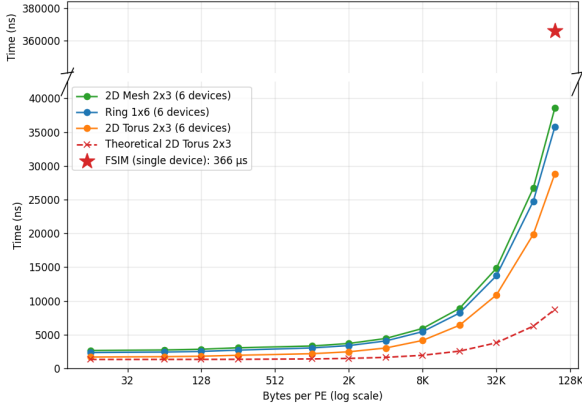


Figure 3: All-reduce latency vs. per-PE payload for the three topologies, against the analytic torus model and an external full-system simulator (FSIM) reference. The measured torus tracks the analytic curve within a small constant factor; the FSIM single-device point (366 μ s) sits an order of magnitude above the KernBench algorithmic latency, illustrating the difference between an achievable-kernel number and a full end-to-end-stack number.

4.3 Results

Following the milestone-evaluation convention, the collective sweep builds its own six-device (six-SIP, 2×3) configurations—distinct from the two-SIP default of Table 1—and measures all-reduce latency as a function of payload size for three inter-device topologies: a 1D ring, a 2D mesh (no wrap), and a 2D torus. Table 2 and Figure 3 report the result.

Three findings follow. First, topology matters and the effect grows with size: at 96 kB/PE the 2D torus is 25 % faster than the mesh and 19 % faster than the ring, because its wrap-around links shorten the worst-case reduction path. Second, the measured curves grow smoothly with payload and stay within a small constant factor of the analytic torus model, with the gap reflecting real link serialization that the analytic model idealizes away. Third, where the IPCQ staging buffer lives is a first-order knob: keeping it in on-PE TCM is materially faster than HBM or shared SRAM at large payloads (Figure 4).

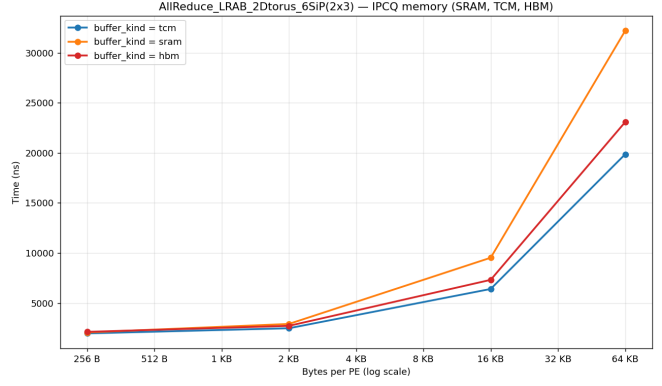


Figure 4: Effect of IPCQ staging-buffer placement (2D torus). At 64 KiB/PE, TCM staging (19 865 ns) beats HBM (23 081 ns) by ~14 % and SRAM (32 201 ns) by ~38 %; at small payloads the three are indistinguishable.

4.4 Analysis and meaning

PE_IPCQ turns the collective from an off-device, contention-prone operation into an on-device primitive whose latency tracks the interconnect’s physical limits. The control/data split keeps the engine small while reusing PE_DMA for movement, and the virtual-channel split is what lets a reduction proceed without stalling the compute DMA that feeds the very GEMMs producing the partials—the same property the fused attention kernel relies on when it interleaves KV reduction with score computation (§5). For the hardware roadmap the results argue two things: provisioning wrap-around (torus) inter-device links is worth roughly a 20–25 % collective-latency reduction at scale, and giving the IPCQ a fast on-PE staging buffer (TCM) rather than forcing it through HBM or SRAM is worth another 14–38 % at large payloads.

5 Fused Grouped-Query Attention

5.1 Why it is needed

Attention is the 1H focus, and it is where the two preceding optimizations have to come together. Grouped-Query Attention (GQA) shrinks the KV cache by sharing each KV head across a group of query heads (here $h_q = 8$ query heads to $h_{kv} = 1$ KV head, a group factor $G = 8$), which makes decoding feasible at long context but also makes it acutely memory-bound: a decode step processes a single query position ($T_q = 1$) against the entire KV history, so its arithmetic intensity is low and its time is dominated by streaming the KV cache out of HBM. FlashAttention-style tiling with an online-softmax merge avoids ever materializing the full score matrix, but realizing it as a fast *fused* kernel needs both building blocks from this report: efficient GEMM issue (§3) for the $Q \cdot K^T$ and $P \cdot V$ products, and an efficient on-device reduction

Table 3: Fused GQA per-panel latency and operation mix. Compute (GEMM, MATH) is a tiny fraction of DMA occupancy; IPCQ copies grow with users and PEs.

Panel	Lat. (ns)	GEMM	IPCQ	DMA
prefill C=1	445	2	0	
prefill C=4 (Ring)	4630	32	24	
decode C=1, P=8	3632	16	21	
decode C=4, P=8	6693	64	93	

(§4) for the multi-user and sequence-parallel KV reductions. This section is the capstone: the fused kernel that uses the composite command and PE_IPCQ at the same time. Multi-head attention (MHA) was studied in prior work and serves here as the established baseline rather than being re-derived.

5.2 Design

The fused GQA kernel issues its matrix products as scheduler-managed composite commands and keeps the online-softmax merge and the cross-device KV reduction inside the kernel, on PE_IPCQ. Two kernel families cover the two phases. The *prefill* kernel is head-parallel and rotates the KV shards around an inter-CUBE ring (“Ring KV”). The *decode* kernel is head-replicated with a statically sharded KV cache and reduces partial attention outputs through an M-fold intra-CUBE chain and, for multiple users, a two-level reduce-to-root. Two further primitives make long context practical: a *lazy load* that issues the KV DMA_READ and returns immediately, auto-waiting only at first use so that KV load overlaps score computation; and per-tile *scratch recycling* that keeps the running softmax accumulators (m, ℓ, O) in a persistent arena while freeing per-tile temporaries, so the kernel fits the 1 MiB scratch budget across many tiles. A further refinement that restructures the decode step into two stateful composites (a named *softmax_merge* recipe) is designed but not yet wired into the measured path; results below reflect the implemented kernel only.

5.3 Results

We measure four headline panels that vary the user count C and the phase: single- and multi-user prefill ($T_q = 4$, $S_{kv} = 16$), and single- and multi-user decode ($P = 8$ PEs, $S_{kv} = 64$ and 128), all at $d_{\text{head}} = 64$ and $G = 8$. For each panel we harvest end-to-end latency (max event end minus min event start, the same window convention as the GEMM study) together with the per-engine busy time and the operation mix. Figure 5 and Figure 6 report the result; the underlying numbers are in Table 3.

The dominant observation is in Figure 6: the compute engines are almost idle. The GEMM engine accumulates only 2 ns to 33 ns of busy time across the panels and the vector-math engine 5 ns to 688 ns, while the DMA engine

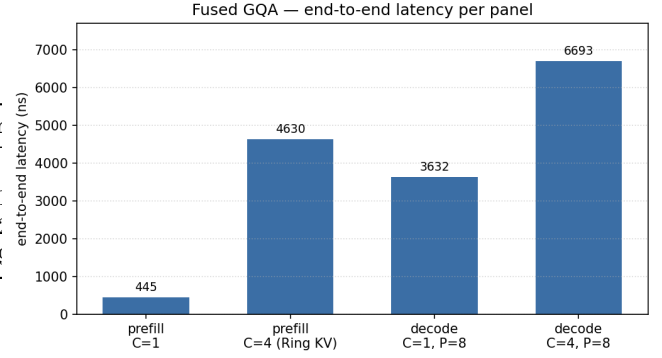


Figure 5: Fused GQA end-to-end latency. Latency grows from 445 ns (single-user prefill) to 6693 ns (four-user decode) as the KV history and the number of participating devices grow.

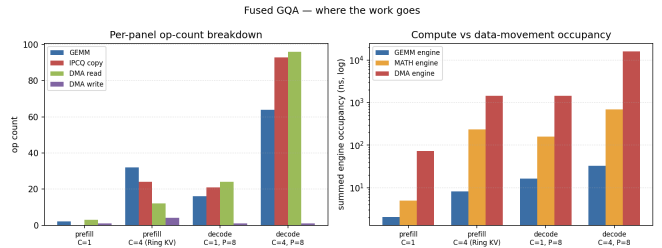


Figure 6: Where the work goes. Left: operation counts—GEMM and IPCQ-copy volume both scale with users and PEs. Right: summed engine occupancy on a log scale—the DMA engine dominates by two to three orders of magnitude over the GEMM and MATH engines in every panel.

accumulates 72 ns to 15 920 ns. Fused GQA, as modeled here, is overwhelmingly data-movement bound. The operation mix shows why the collective machinery matters: IPCQ-copy count rises from zero (single-user prefill) to 93 (four-user decode) as the kernel reduces partial outputs across more PEs and CUBEs, and DMA-read count rises in step as more KV shards are streamed. The PE control-processor dispatch cost registered as zero in this configuration—command issue is simply not on the critical path when data movement is this dominant.

5.4 Analysis and meaning

These panels are the clearest statement of the code-sign thesis in the report. Because the composite command keeps GEMM issue cheap and the MAC array barely occupied, the fused attention kernel’s latency is set almost entirely by data movement: streaming the KV cache and reducing partials across devices. That is precisely the cost that the communication-side work targets—PE_IPCQ for the on-device reduction, the lazy load for load/compute overlap, fast TCM staging and torus links for the reduction itself. In other words, the two enablers are not independent features that happen to appear in the same kernel; the GEMM optimization is

what *exposes* the data-movement bottleneck (by removing the compute and issue overhead that would otherwise hide it), and the communication optimization is what *attacks* it. For an attention-dominated decoder the meaningful hardware investments are therefore the ones that move data faster and reduce it on-device—not additional MAC throughput, which this workload cannot use.

6 Discussion: which hardware changes are meaningful

Read together, the three studies point to a consistent ranking of where hardware investment pays off for attention-centric decoding.

The composite command is foundational, but indirectly. Its direct effect—driving compute-rich GEMMs to ~78% of peak—matters most for the compute-bound parts of a model (the large feed-forward and projection matrices). For attention itself, its more important effect is *diagnostic*: by making issue and compute nearly free, it removes the overhead that would otherwise mask the true bottleneck, and the fused GQA results then show unambiguously that the kernel is data-movement bound. Without a cheap, self-routing issue mechanism we would not be able to tell whether attention is slow because of compute or because of data movement; with it, the answer is clear.

The communication path is where attention latency actually lives. Every all-reduce and fused-GQA measurement says the same thing: the limiting resource is moving and reducing data, not multiplying it. That makes the PE_IPCQ design and the choices around it the highest-value hardware levers for this workload:

- **On-device collectives with a compute/communication virtual-channel split.** Performing the reduction on the device, with `vc_comm` separated from `vc_compute` so the reduction does not stall the compute DMA, is what lets attention overlap KV movement with score computation at all.
- **Fast on-PE staging memory.** Keeping the IPCQ buffer in TCM rather than HBM or SRAM is worth 14–38% of collective latency at large payloads—a pure placement decision with a first-order effect.
- **Wrap-around (torus) inter-device links.** A torus fabric buys 20–25% over a mesh or ring at scale by shortening the worst-case reduction path.

Raw MAC throughput is not the constraint for attention. The GQA panels leave the GEMM and vector-math engines two to three orders of magnitude below the DMA engine in busy time. Adding MAC area would not

move decode latency; the workload cannot use it. This is the single most actionable finding for an attention-dominated roadmap.

Caveats. These conclusions are achievable-kernel results from a deterministic model, not E2E measurements; absolute numbers carry the model’s idealizations (§2.3), though the relative rankings that drive the recommendations are robust to them. The headline GQA panels are also at modest scale (up to four users, single SIP), and one designed refinement—the two-composite `softmax_merge` decode—is not yet in the measured path. Larger-scale and multi-SIP headline runs are 2H work.

7 Conclusion

This 1H work set out to make attention-centric LLM kernels fast through hardware–software codesign, and to do so on a platform that isolates algorithm-level behavior from the rest of the software stack. The result is a coherent picture rather than three separate optimizations. A composite command that issues a tiled GEMM as one self-routing pipeline makes the MAC array usable—reaching ~78% of peak on compute-rich shapes with measured efficiency tracking theory—and, just as importantly, makes compute cheap enough that the real bottleneck becomes visible. A per-PE on-device collective engine, PE_IPCQ, turns all-reduce into a primitive whose latency follows the interconnect’s physical limits, with topology and staging-memory choices each worth tens of percent. Fused Grouped-Query Attention then combines the two and shows the payoff and the lesson at once: the kernel is data-movement bound, so the optimizations that move and reduce data—not those that add arithmetic—are what determine its speed.

The practical conclusion for the hardware roadmap is therefore specific. The changes worth keeping are the single-command self-routing GEMM pipeline, the on-device PE_IPCQ collective with its compute/communication virtual-channel split, fast on-PE staging memory, and wrap-around inter-device links. Additional MAC throughput is not, for this workload, a meaningful investment. KernBench made these conclusions measurable by holding everything except the algorithm and the hardware fixed; the next half extends the same method beyond attention to the rest of the decoder.

8 Future Work — 2H

The 1H work covered attention end to end. The natural next step is to complete the decoder and then to ask how compute and data should be distributed for realistic, agentic workloads.

From attention to the full decoder: FFN and MoE. A decoder block is attention followed by a feed-

forward network (FFN), and in modern models that FFN is increasingly a mixture-of-experts (MoE) layer. The FFN is the compute-rich counterpart to attention—it is where the composite command’s MAC-efficiency gains (§3) should matter most—so adding it gives a balanced view of a full block instead of its memory-bound half alone. MoE adds a new dimension: a routing step selects a few experts per token, turning the dense FFN GEMM into a sparse, data-dependent dispatch. The open questions are how to issue expert GEMMs as composites under data-dependent token counts, and how to move tokens to experts efficiently—an all-to-all-shaped communication pattern distinct from the all-reduce studied here, and a natural extension of the PE_IPCQ work.

Compute and data distribution for full LLM decoding. With both attention and FFN/MoE in hand, the question becomes where each layer’s compute and state should live. Attention is KV-bound and favors keeping the KV cache close to the PEs that consume it; FFN/MoE is compute-bound and favors spreading GEMM work across PEs; MoE routing makes the optimal placement token- and time-dependent. A 2H goal is to use KernBench to explore these placement and parallelization trade-offs—tensor vs. expert vs. sequence parallelism—under a single, software-stack-independent model, so the interconnect and memory implications of each choice are measured rather than assumed.

Agentic workloads. Agentic inference interleaves many short, bursty decode requests with tool use and long shared contexts, which stresses the system differently from a single long generation: context reuse across requests, dynamic batching, and uneven expert load all change how compute and data should be dispersed. Characterizing how total compute and data movement distribute across the SIP/CUBE/PE hierarchy under such workloads—and which of the 1H hardware levers still dominate when the workload is this irregular—is the broader 2H agenda.