

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

Mukesh Garg

Jiyeon Lee

Yangwook Kang

AGI Computing Lab, System Technology Group
2026 H1 Report

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. Looking ahead, these same mechanisms provide the basis for optimizing FFN-intensive workloads such as Mixture-of-Experts (MoE) layers and for developing integrated optimization strategies spanning both attention and MoE components within next-generation AI models.

1 Introduction

AHBM integrates compute units directly into the HBM stack. Each processing element (PE) is paired with a dedicated slice of HBM and uses local TCM and SRAM to stage data between memory and the MAC array. On this memory-centric architecture, kernel performance depends not only on compute throughput but also on how effectively data is placed, moved, and shared across the memory hierarchy and between PEs. Optimizing AI kernels on AHBM therefore requires hardware–software co-design, in which kernel algorithms and architectural mechanisms are developed together.

To enable detailed performance analysis and rapid design exploration, we developed **KernBench**, a source-level discrete-event simulation platform for AHBM. KernBench implements the AHBM execution model, including memory-system latencies, the PE execution model, inter-PE communication, and host-side orchestration, while executing both kernel and host software directly from source code. This provides fine-grained visibility into execution behavior. It enables systematic evaluation of hardware–software co-design choices independently of higher-level software stacks such as compilers and runtimes. All results presented in this report are obtained using KernBench, which serves as the common evaluation platform for all mechanisms and kernels discussed in this study.

This report focuses on Grouped-Query Attention (GQA), one of the most performance- and bandwidth-critical components of LLM inference. GQA dominates inference-time memory traffic and KV-cache capacity in modern LLM serving, making it the primary bandwidth bottleneck on memory-centric architectures such as AHBM. Modern decoder-only models such as Llama 3 and Mistral have largely transitioned from GPT-3-style multi-head attention (MHA) to GQA, in which multiple query heads share a single KV head to reduce KV cache capacity and memory-bandwidth requirements. While GQA improves system efficiency at the model level, mapping it 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. First, GQA-aware data placement distributes KV caches and weights across the TCM/SRAM/HBM hierarchy to reduce communication overhead and improve data locality. Second, PE_IPCQ provides an efficient on-device collective communication primitive for reductions and other communication-intensive operations. Third, a composite-command GEMM pipeline tightly pipelines memory movement and computation within each PE under PE_SCHEDULER control, reducing command overhead while keeping the MAC array efficiently utilized.

The compute enabler (the composite-command GEMM pipeline, §3) and the communication enabler (PE_IPCQ, §4) are first developed and evaluated independently. The fused GQA kernel (§5) then combines them with GQA-aware data placement to demonstrate an end-to-end attention implementation on AHBM. The correspondence is direct: attention’s $QK^\top$ and PV products are precisely the GEMMs that benefit from the composite-command pipeline, while its KV reductions are precisely the collective operations that benefit from PE_IPCQ. Together, these mechanisms enable the fused GQA kernel to efficiently exploit AHBM’s HBM bandwidth.

Although GQA serves as the motivating workload for this study, the resulting mechanisms are intended as reusable building blocks for a much broader class of AI kernels. PE_IPCQ can support collective communication across distributed and communication-intensive workloads, while composite-command execution can be applied to GEMM-based kernels, feed-forward networks (FFNs), normalization, and other fused operator pipelines. Together with the hierarchical data-placement framework, these mechanisms form a foundation for future AI kernels and communication libraries on AHBM. In the second half of 2026, this foundation will be extended to FFN- and MoE-dominated workloads and to end-to-end optimization of complete LLM execution.

The remainder of this report is organized as follows. Section 2 describes the KernBench platform and the AHBM configuration used throughout this study. Sections 3, 4, and 5 present the composite-command GEMM pipeline, PE_IPCQ collective communication, and the fused GQA kernel, respectively. Section 6 discusses the broader architectural implications of these results. Finally, Sections 7 and 8 summarize the key findings and outline future work on FFN, MoE, and full-model optimization.

2 The KernBench Platform

All results in this report are produced on *KernBench*, a system-level discrete-event simulator for LLM kernels running on AHBM. This section explains why the platform exists, the device and execution model it presents to a

kernel writer, how its latency model turns that execution into a number, and the concrete hardware configuration used for every experiment that follows.

2.1 Why KernBench

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?*

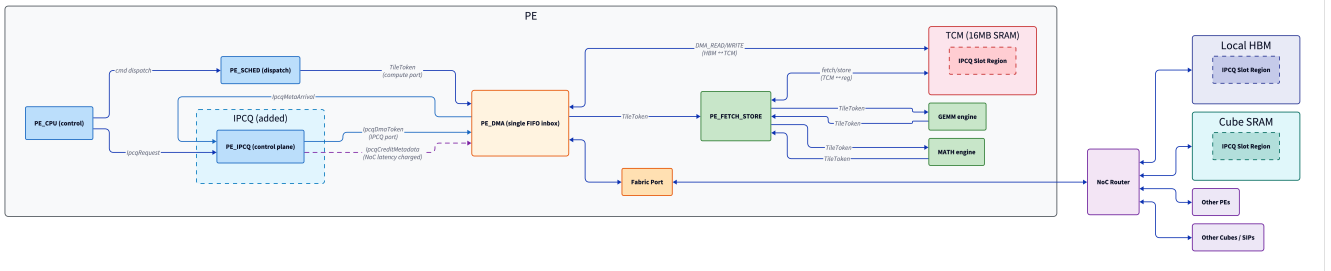
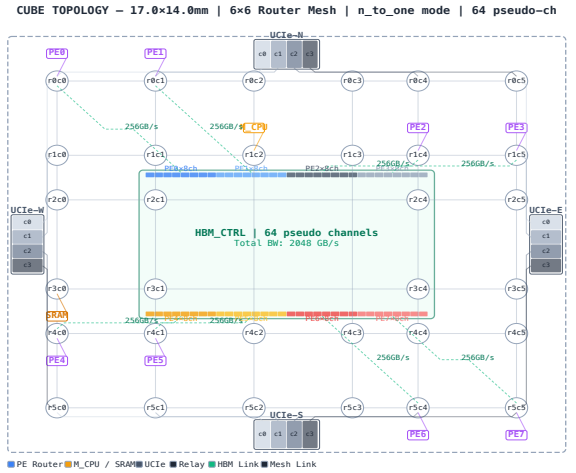
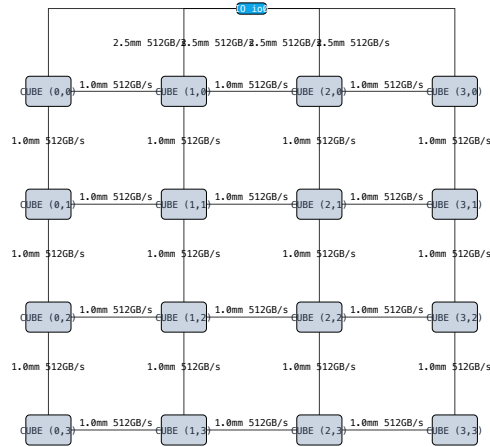
KernBench is built to answer exactly that question. Kernels are written and executed at the *source level*—as algorithmic descriptions in a small tile-oriented kernel API—with no dependency on a compiler or any other software-stack layer. The simulator takes the kernel and a hardware topology and reports the latency 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 an ideal software stack would expose.

2.2 Device and execution model

KernBench models AHBM as a hierarchy of SIPs, CUBEs, and processing elements (PEs). At the system level, multiple SIPs are connected through inter-package links, while each SIP contains a collection of CUBEs joined by an on-package interconnect (Fig. 1a).

Each CUBE contains eight PEs, shared SRAM, HBM controllers, an M_CPU control processor, and an intra-CUBE router mesh (Fig. 1b). Together these components form the execution substrate for all kernels evaluated in this report. This organization reflects the memory-centric nature of AHBM: each PE is paired with a dedicated slice of HBM bandwidth and local on-PE storage (TCM), so compute lives next to the data it consumes rather than fetching it through a far-away memory controller. The platform’s performance question is therefore not “how many FLOPs can the chip do” but “how well can a kernel keep each compute engine fed from its locally-attached memory while moving the unavoidable traffic between PEs efficiently.”

Within a PE, commands are dispatched by PE_CPU to PE_SCHED, which routes work to specialized execution engines—DMA, FETCH/STORE, GEMM, vector-math, and IPCQ (Fig. 1c). Commands come in two flavours. *Atomic* commands target a single engine—a plain DMA read, a GEMM tile, a vector-math op, an



(c) PE level (zoom-in of one PE in (b)): PE_CPU dispatches commands to PE_SCHED, which routes tile-token streams through PE_DMA, PE_FETCH_STORE, and GEMM/MATH engines; PE_IPCQ is the on-device collective control plane.

Figure 1: Modeled hardware graph at the SIP, CUBE, and PE levels. This figure is an *illustrative topology* chosen for readability; the *experimental configuration* used throughout this report (port counts, slice counts, and link parameters) is specified in §2.5 / Table 2, and numbers in the two may differ. KernBench is not tied to this particular arrangement: each box is a modeled component node, each line a directed link with bandwidth and propagation attributes, and any topology that respects those attributes is supported.

IPCQ send/receive—and are the natural unit for short or special-purpose work; the PE_CPU itself also runs control-plane work directly. *Composite* commands, by contrast, carry an ordered pipeline of operations across multiple engines (DMA_READ $\rightarrow$ FETCH $\rightarrow$ GEMM/MATH $\rightarrow$ STORE $\rightarrow$ DMA_WRITE) that the PE_SCHED tiles and streams without per-tile redispach. The composite form is the substrate for the GEMM optimization (§3) and, combined with on-PE collectives, for fused attention (§5).

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 and routes every request through the modeled graph.
- The **components** are device-side nodes that model hardware behaviour: the per-PE blocks (scheduler, DMA, GEMM and vector-math engines, TCM, IPCQ), the NoC routers, the HBM controllers, and the inter-chiplet links.

Data and timing are handled in two passes, so that a kernel’s numeric results and its latency are computed consistently but independently. **Pass 1 (timing)** runs the kernel under the discrete-event engine: memory ops (`tl.load`, `tl.store`) execute against a host-side `MemoryStore` and return real tensor data, while compute ops (`GEMM`, `vector-math`) only emit records into an op-log carrying their operands, shapes, dtypes, and scheduled time. **Pass 2 (data)** replays that op-log offline in `numpy`, producing the actual numeric outputs and comparing them against a reference computation under per-dtype tolerances (e.g. `rtol/atol = 10-3` for `f16`). Pass 2 is optional—runs that need only latency skip it—but when enabled it guarantees that every reported timing number corresponds to a kernel whose numeric output has been verified end-to-end.

2.3 Latency model: graph traversal and contention

The modeled hardware hierarchy described above is represented internally as a directed graph. Nodes corre-

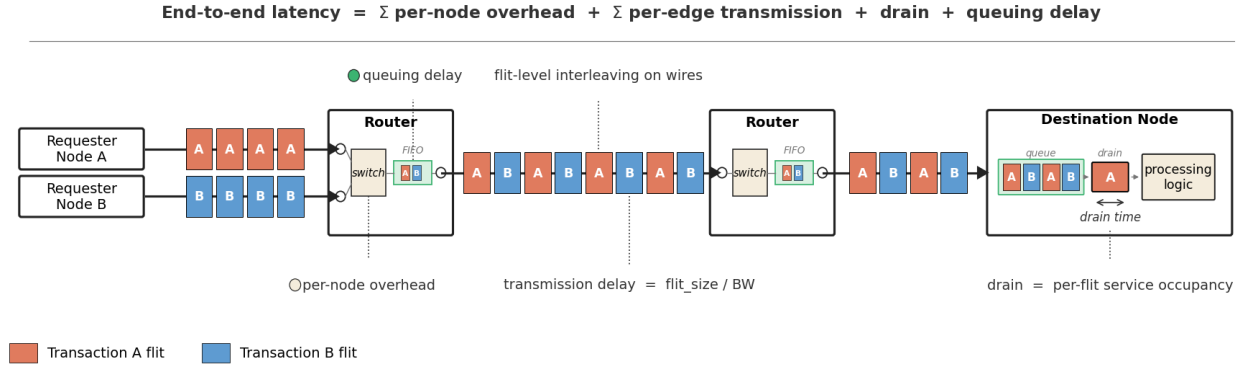


Figure 2: Conceptual schematic of the latency model. Two source nodes (Requester A/B) inject flits through a chain of routers into a destination node; on the shared edge between routers, flits from the two transactions are interleaved flit-by-flit by the wire’s FIFO arrival order. End-to-end latency is the sum of four contributions: **per-node overhead** (the switch’s fixed processing cost, shown in light yellow—the same colour as the destination’s processing-logic block), **per-edge transmission** ($\text{flit_size}/\text{BW}$ on each wire), **drain** (per-flit service occupancy at the destination’s channel), and **queuing delay** (waiting in a FIFO when a shared resource is busy). The places where queuing actually accumulates are highlighted in green: the router output queue and the destination’s input queue. This is the model, not a measurement; specific bandwidths and overheads are listed in §2.5 (Table 2).

spond to hardware components—PEs, routers, memory controllers, SRAM blocks, IO chiplets—while edges represent communication links with associated bandwidth (GBs^{-1}) and propagation (ns) attributes. Every operation in KernBench—DMA transfers, remote memory accesses, collective communication, command dispatch—is modelled as a traversal through this graph. End-to-end latency is decomposed into four contributions accumulated along the traversal path (Fig. 2): **per-node overhead** at each component, **per-edge transmission** on each wire, **drain** (per-flit service occupancy) at the destination, and **queuing delay** at the shared FIFOs that the wire and the destination share between concurrent transactions.

The hardware as a graph. The topology is compiled once at configuration time into this graph and is never mutated during a run. There are no hidden shortcuts, implicit bypasses, or magic paths: if a request reaches its destination, the path it took is explicit in the graph, and the latency it incurred is the sum of the per-node and per-edge costs paid along that path. The same graph representation applies recursively at every hierarchy level—system, SIP, CUBE, and PE (Fig. 1).

From graph to discrete-event simulation. The graph is driven by a discrete-event engine. Two kinds of events advance simulation time: *node events* (component switching overhead, service completions such as an HBM channel commit or a GEMM tile finish) and *edge events* (the flit-by-flit serialization of a payload across a bandwidth-limited link). The engine maintains a priority queue of pending events ordered by their scheduled time and fires them one at a time, with ties broken under a deterministic policy so that the same kernel on the same

topology always yields the same trace. Per-request correlation IDs are stamped at injection and carried through every hop, so the path from injection to completion is fully traceable. Every modeled latency contribution corresponds to exactly one of these events on exactly one node or edge—there is no slack in the budget.

Latency contributions. Three kinds of latency accumulate along a traversal: (i) *per-node fixed overhead*—each component carries a small switching cost (router decode, controller pickup, scheduler handoff); (ii) *per-edge transfer time*—each link’s payload is decomposed into fixed-size flits (default 256 B), and each flit arrives at $\text{prop} + \text{flit_bytes}/\text{bw}$ after the previous one, giving wormhole semantics across multi-hop paths; and (iii) *per-service occupancy*—memory controllers, GEMM stages, and collective engines hold the request for their service time before releasing it downstream. Each of these is attached to a specific node or edge in the graph; together they make up the entire latency budget.

Beyond data movement and execution latency, KernBench also models the control-plane cost required to issue work to accelerator engines. Since every DMA, GEMM, vector-math, and IPCQ operation is initiated through the $\text{PE_CPU} \rightarrow \text{PE_SCHED}$ path, command dispatch latency contributes to the end-to-end execution time of all kernels.

Command dispatch overhead model. The cost incurred by the PE_CPU when it dispatches a command to one of the accelerator engines is modelled structurally rather than with a per-operation calibration table. The PE_CPU charges, per command,

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

This linear-in-size form reflects the serialization cost of writing a command descriptor into the scheduler queue: a fixed per-command bookkeeping cost plus a byte-wise descriptor-transfer cost. Concretely, 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. This `FIXED` is distinct from Table 2’s `2 ns / 1 ns PE_CPU / PE_SCHED` fixed costs: the latter are per-component traversal overheads each command pays when it transits those nodes, whereas the `FIXED` term here is the command-descriptor issue cost. `FIXED` depends on the command class: a single-op command — one engine operation, i.e. a single DMA descriptor, GEMM, or element-wise issue — carries a lighter 8-cycle `FIXED`, while the composite command, which the scheduler expands into a multi-stage tile-feeder plan, carries 40 cycles (§3.4). With the composite `FIXED` = 40 cycles and $R = 0.0625$ cycles/byte (i.e. 16 B cycle^{-1} , at 1 GHz) a typical composite lands 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. 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.

2.3.1 Congestion and contention modeling

The simulator’s sharpness comes from how it models contention for those nodes and edges. *Every directed edge has a FIFO*: an arriving flit takes its bandwidth-limited transfer time on top of whatever earlier flits are still being served, so a busy link queues later traffic behind earlier traffic rather than transferring everything at peak BW. *HBM is modelled with per-pseudo-channel parallelism*: a stateless array of channel-availability timestamps with address-based channel selection captures the bank-level concurrency that real HBM exposes, so 64 channels per CUBE deliver real parallelism on uniform addresses but a hot channel surfaces as the bottleneck on skewed ones. *Every component has a serial worker*: a router carrying two heavy streams interleaves them at flit granularity in arrival order rather than fanning out for free, so two concurrent collectives sharing a link share its bandwidth, not double it. Without these mechanisms the simulator would simply re-confirm the peak-BW roofline; with them, it reveals where the real bottlenecks form and which hardware levers actually relieve them—which is exactly the question the codesign work in this report turns on.

2.4 Accuracy

The model is precise about the effects that dominate kernel latency on this class of hardware: per-edge bandwidth occupancy and flit-level serialization, HBM pseudo-channel parallelism, and per-component switching over-

head. Two independent cross-checks drawn from the experiments in this report confirm that this precision translates into physically reasonable kernel latencies.

First, in the GEMM study (§3), simulator-measured MAC efficiency tracks an analytic ideal-pipeline model within 1.4 ppt across every swept shape—from a single-tile $M=K=N=32$ at $\sim 7.7\%$ up to the deep- K $K=3072$ case at $\sim 90\%$. The residual gap is fill/tail overhead the closed-form pipeline model omits.

Second, in the all-reduce study (§4, Fig. 11), simulator latency for a 2D-torus over six devices follows the expected startup-plus-per-packet shape across the entire payload sweep—tight at small payloads where startup dominates, and within a single-digit multiplicative factor at the largest payloads, where the residual gap is explained by per-router switching the analytic shape elides.

Third, the simulator’s per-traversal behaviour can be probed directly. Running `kernbench probe` on the modelled topology issues a sequence of PE-to-HBM DMA reads at progressively greater hop distances and reports the per-component overhead, per-edge serialization, and per-PC drain that the model charges (Table 1). Three properties stand out. First, the reported latency increases *monotonically* with hop count—from 141 ns at the local HBM slice to 677 ns at the worst-case remote-CUBE slice—with the increment per added hop matching the per-router overhead and the UCle-link cost in Table 2. Second, the bandwidth-saturation curves track the wire’s serialization model: at 1 MiB transfers the local PE DMA reaches 100% of the per-edge bandwidth limit, while the longest cross-CUBE path saturates at 97.8%—exactly the gap that the per-edge propagation and per-router overhead model would predict. Third, the simulator passes the internal-consistency invariants the probe builds in (monotonic hop progression; D2H reads at least as long as the equivalent H2D writes because of the reverse-path acknowledgement; cross-CUBE best less than cross-CUBE worst). These are properties that hold *only* when the underlying graph traversal, FIFO contention, and propagation models are internally self-consistent—a model error in any of them would surface as a non-monotonic or under-utilising curve here long before it polluted a kernel-level measurement.

The known simplifications—FIFO router arbitration (instead of round-robin), HBM scheduler without write-buffer reordering, no bank conflict, no refresh or thermal effects, and no upstream backpressure—are the price of a deterministic, inspectable model. These simplifications bound the absolute accuracy, but the agreement with both analytic models and the probe’s internal-consistency checks above indicates that KernBench is sufficiently accurate for evaluating the *relative* hardware–software design trade-offs (tiling A vs. B, topology X vs. Y, with vs. without composite command, mesh vs. torus) that are the primary objective of this work.

Table 1: PE $\rightarrow$ HBM DMA latency probe at varying hop distances (32 KiB transfer; output captured from `kernbench probe`). *Util%* is the achieved bandwidth as a fraction of the path’s bottleneck-edge bandwidth.

Case	Latency (ns)	Util% (32 KiB)	Util% (1 MiB)
PE $\rightarrow$ local HBM	141.0	90.8	100.0
PE $\rightarrow$ same-half HBM	147.9	86.6	99.9
PE $\rightarrow$ cross-half HBM	161.2	79.4	99.8
PE $\rightarrow$ cross-CUBE (best)	330.5	77.5	99.6
PE $\rightarrow$ cross-CUBE (worst)	677.1	37.8	97.8

2.5 Modeled hardware configuration

Table 2 summarizes the hardware configuration used throughout this report. The intent of this configuration is not to model a specific product, but to represent a realistic memory-centric accelerator and to provide a consistent baseline for evaluating hardware– software co-design mechanisms.

The compute capability within each CUBE is provisioned such that inference workloads can effectively saturate the available HBM bandwidth. The aggregate compute throughput is therefore balanced against memory-system bandwidth rather than being intentionally over- or under-provisioned. Concretely, 8 PEs $\times$ 8 TFLOP s^{−1} give roughly 64 TFLOP s^{−1} of f16 compute per CUBE against 2048 GB s^{−1} of HBM bandwidth, which places the balanced point at about $\sim$ 31 FLOP/byte; inference decode kernels typically operate well below that, so HBM bandwidth—not compute—is the natural ceiling on this configuration. For reference, the GQA decode kernels evaluated in §5 operate at arithmetic intensity well below this balance point, so their ceiling is set by HBM and inter-PE traffic rather than by GEMM throughput. HBM bandwidth is distributed evenly across the PEs within a CUBE, with each PE responsible for servicing approximately one-eighth of the CUBE’s memory bandwidth through its dedicated HBM channels.

The on-chip interconnect is configured using bandwidth and latency parameters representative of commercially available mesh-network IPs. The goal is not to study a particular NoC implementation, but rather to evaluate kernel behavior under a realistic baseline communication fabric.

For die-to-die communication, UCIE-A bandwidth characteristics are used as the reference point for inter-CUBE links. Inter-SIP communication is modeled using PCIe-class links. Together, these assumptions provide a representative communication hierarchy for evaluating collective communication and distributed kernel execution.

Unless otherwise stated, all experiments use this configuration. Workload-specific parameters such as matrix dimensions, sequence lengths, and collective payload sizes are introduced in their respective sections.

Table 2: Modeled hardware configuration

Parameter	Value
<i>Hierarchy</i>	
SIPs	2 (1D ring)
CUBEs per SIP	16 (4 $\times$ 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
PE_CPU fixed cost	2 ns
PE_SCHED fixed cost	1 ns
<i>Memory (per CUBE)</i>	
HBM capacity	48 GB (8 slices)
HBM aggregate BW	2048 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 single-op command	8 cycles
FIXED per composite command	40 cycles
per-byte rate <i>R</i>	0.0625 cycles/byte (16 B cycle ^{−1})
composite size cap	1024 B

3 GEMM Acceleration via the Composite Command

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?

3.1 Design

The answer is the *composite command*. A single command carries the ordered five-stage tile pipeline (DMA_READ, FETCH, GEMM, STORE, 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 through the chain 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).

3.2 Results

We sweep eight GEMM shapes spanning square, tall, wide, and deep- K geometries under two operand-staging variants that bracket the realistic LLM cases. In *load_ref*, the activation A is pre-staged on chip and only the weight W streams from HBM during the composite (the “activation in TCM, weights from HBM” case typical of decoding with a small batch). In *ref_ref*, both operands stream from HBM during the composite (the “cold both sides” case, where the activation does not fit on-chip or is itself the output of a producer composite that wrote back to HBM). Both variants run the same composite command; only the up-front staging of A differs.

We evaluate the composite GEMM along two axes that together describe how well it lands on the hardware: *HBM bandwidth utilization* (does the workload saturate the per-PE HBM bandwidth, i.e. is it memory-bound on this configuration?) and *achieved GEMM throughput* (what fraction of the 8 TFLOP s^{-1} per-PE peak does the kernel actually deliver). Both metrics use the composite window as the denominator, so the up-front tl.load of A in *load_ref* is excluded — only HBM traffic and compute that happen *inside* the composite count.

The composite reaches the hardware roofline in two distinct regimes. *Memory-bound*: $K=3072$ deep- K saturates HBM (88% *load_ref*, 94% *ref_ref*) and the low-

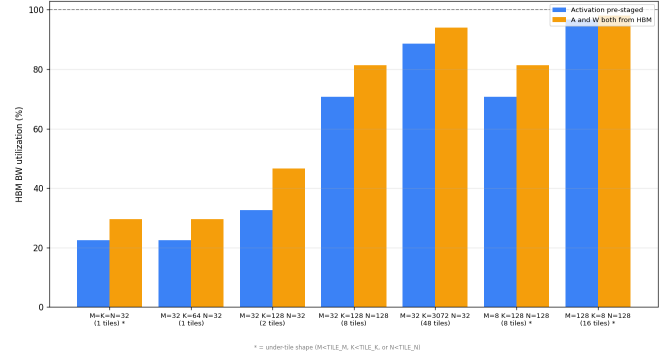


Figure 3: Per-PE HBM bandwidth utilization during the composite window, for the two staging variants. The ceiling is the per-PE HBM bandwidth (256 GB s^{-1} ; dashed line at 100%). *ref_ref* streams both A and W and so always sits at a higher BW-utilization than *load_ref* on the same shape; both variants saturate ($> 85\%$) once the kernel has enough total bytes to keep the link busy (deep- K $K=3072$, or low-reuse output-dominated $M=128, K=8, N=128$). Small or single-tile shapes do not have enough total traffic to saturate the link and are bottlenecked by pipeline fill rather than HBM.

reuse $M=128, K=8, N=128$ corner saturates even harder ($> 96\%$ in both), because back-to-back output writes dominate. *Compute-rich*: at the same deep- K shape, *load_ref* delivers 7.18 TFLOP s^{-1} ($\sim 90\%$ of the per-PE peak) — the configuration’s clean win. The *distance between the two variants* at the same shape is the cost of *not* pre-staging A : at $K=3072$ the throughput halves (7.18 TFLOP $s^{-1} \rightarrow 3.83$ TFLOP s^{-1}) because the second operand doubles HBM pressure and the kernel hits the BW ceiling. This is the operational take-away — when the activation can live on-chip the composite delivers near-peak GEMM; when it cannot the BW ceiling dominates and throughput halves.

Figure 5 validates that simulator-measured efficiency tracks an analytic ideal-pipeline model (within 1.4 ppt across every swept shape), and Figure 6 resolves the per-stage engine wall-clock that backs the above interpretation.

3.3 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 the issue cost once per GEMM rather than once per tile-stage, and by chaining tile-stages through token self-routing, it lets the scheduler keep every stage busy on a different tile, so compute-rich shapes actually reach the efficiency their arithmetic intensity allows ($\sim 90\%$ of GEMM peak at the deep- K *load_ref* corner) and data-bound shapes actually reach their HBM-BW ceiling instead of stalling on command overhead.

The split between *load_ref* and *ref_ref* also makes the hardware-software boundary explicit: when the ac-

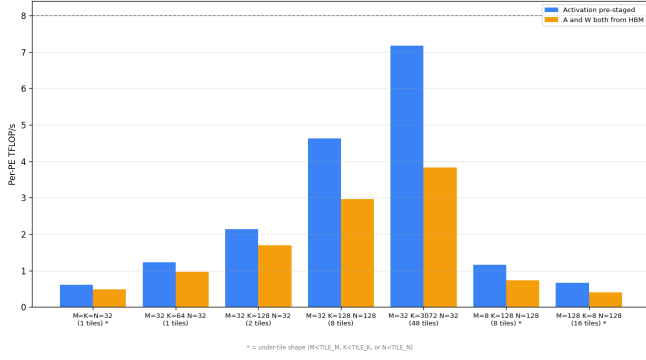


Figure 4: Per-PE GEMM throughput delivered during the composite window (reference line at the 8 TFLOP s⁻¹ per-PE engine peak). At the deep- K corner $M=32, K=3072, N=32$ the activation-pre-staged kernel reaches 7.18 TFLOP s⁻¹ ($\sim 90\%$ of peak); the both-from-HBM variant drops to 3.83 TFLOP s⁻¹ ($\sim 48\%$) at the same shape because the second operand doubles HBM pressure and the kernel hits the BW ceiling shown in Fig. 3. Under-tile shapes ($M=K=N=32$, $M=8, K=8$) are hard-capped by tile-fill at 50%, 25%, 12.5% of peak regardless of staging.

tivation fits in on-chip storage the composite is BW-headroom-rich and saturates the GEMM engine; when it does not, both operands compete for the same HBM port and the kernel is BW-bound. This is the lever the GQA kernel of §5 reaches for next — keeping the right working set on-chip so the composite pipeline lands in the compute-rich regime rather than the BW-bound one.

3.4 Why composite, and not kernel-orchestrated async loading?

A reader familiar with double-buffered GEMM kernels on conventional hardware may ask: why a hardware-side composite command at all? Why isn’t the obvious kernel-level pattern — async-load each operand, overlap with compute, accumulate — sufficient?

To answer this concretely we contrast composite against two kernel-orchestrated baselines that have access to the same single-op primitives the platform exposes (tl.load for async DMA into TCM, tl.dot for a single-op GEMM command on TCM-resident operands, tl.store for a DMA write-back). Both baselines pre-stage activation A identically to load_ref, so the window measured is the engine-pipeline window with A ’s up-front DMA excluded for all three kernels.

Async-full (naive). A single tl.load(A) followed by a single tl.load(B) (async, queued behind A), tl.dot(A, B), and tl.store(out). The kernel issues four commands total. The decisive constraint is that tl.dot’s `_await_pending(b)` blocks the GEMM command until the *entire* B has landed in TCM — there is no way at the runtime API surface to express “start computing on tile 0 of B while tile 1 is still in flight.” Load-of- B and GEMM therefore serialize.

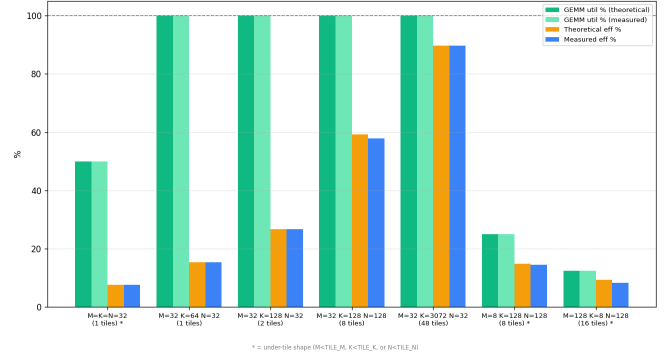


Figure 5: GEMM MAC utilization and efficiency, theoretical vs. measured (load_ref staging). Tile-fill sets the ceiling: under-tile shapes (marked *) such as $M=K=N=32$, $M=8$, and $K=8$ cannot fill the MAC tile and cap at 50%, 25%, 12.5%. For tile-filling shapes, efficiency climbs with tile count—from $\sim 15\%$ at one tile to $\sim 90\%$ measured at 48 tiles ($K=3072$)—and the measured bars track the analytic ideal-pipeline prediction within 1.4ppt across every shape.

Async-tiled (chunked prefetch). The kernel-level workaround is to split B along K into TILE_K-sized chunks, issue async tl.loads for those chunks, issue one tl.dot per chunk (each blocking only on its own b_i), and accumulate via `out = out + tl.dot(...)`. This is the standard double-buffered-GEMM pattern transcribed to the single-op primitives. We measure two versions of this kernel that differ only in *prefetch depth*: **depth-2 (TCM-bounded)** keeps at most two B-chunks in flight at any time by issuing the next tl.load just before each tl.dot; **depth- ∞ (queue-all)** issues all N_K B-chunk loads up front so the DMA engine has the deepest possible request queue. For a $K=3072$ shape both versions emit 48 A -chunk loads + 48 B -chunk loads + 48 tl.dots + 47 elementwise adds + 1 store = 192 host-side commands; the only difference is the temporal interleaving of B-load and dot dispatches.

Why two depths. The depth distinction matters because the async-full kernel and the depth- ∞ async-tiled kernel both pin the *entire* B in TCM simultaneously — $K \cdot N \cdot 2$ bytes. The on-PE scratch is capped at 1 MiB (topology.yaml: `pe_tcm.kernel_scratch_mb=1`), so an LLM-scale attention $B = K_{KV} \times d_{\text{head}}$ at $K_{KV} = 4096, d_{\text{head}} = 128$ already needs 1 MiB, and at $K_{KV} = 8192$ it needs 2 MiB — past the cap. async-full and queue-all async-tiled are therefore not just slower than composite but *architecturally infeasible* at LLM context length. The depth-2 async-tiled kernel is the only kernel-level variant whose peak TCM footprint stays $O(2 \cdot \text{TILE_K} \cdot N)$ regardless of K , the same order as composite’s per-tile streaming buffer. It is the apples-to-apples comparison.

Per-PE throughput. Figure 7 reports per-PE TFLOP/s for all four kernels side-by-side. The single-op fast-path in the dispatch cost model (§2.3.1) is enabled,

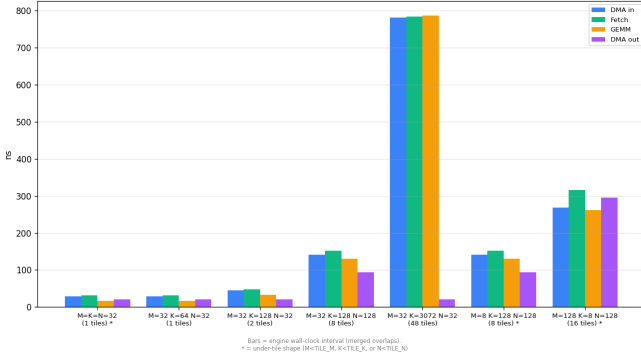


Figure 6: Per-stage engine wall-clock (DMA in, Fetch, GEMM, DMA out) under `load_ref` staging. For the deep- K shape ($K=3072$, 48 tiles) DMA in, Fetch, and GEMM are all close to ~ 785 ns, running concurrently in a tightly pipelined regime while DMA-out is negligible. For the low-reuse shape ($M=128, K=8, N=128$, 16 output tiles) DMA-out grows to ~ 336 ns while GEMM compute is ~ 262 ns—a data-movement-bound regime where the output write becomes the largest stage.

so every single-op command the async kernels emit — DMA descriptors and `tl.dot/tl.add` alike — is charged the light 8-cycle `FIXED`, not the 40-cycle composite control-path cost; neither async-tiled variant carries an inflated per-command cost.

The $K=3072$ corner, concretely. We take this shape ($M=32, K=3072, N=32$) as the running example throughout the mechanism discussion because it is the regime where the four kernels spread the widest — the deepest K in the sweep maps onto $K/\text{TILE_K} = 48$ hardware tiles, so per-tile costs amplify into the largest measurable gap. The work content is identical for all four kernels: ~ 6.3 M f16 MACs and ~ 386 KiB of B traffic from HBM, which together require ~ 786 ns of GEMM-engine compute and ~ 781 ns of DMA on a saturated per-PE link. What differs is the number of host commands the same work is decomposed into — 2 for composite (one `tl.load(A)` plus one composite), 4 for async-full, and 192 for either async-tiled variant (48 A -loads + 48 B -loads + 48 `tl.dots` + 47 elementwise adds + 1 store). The engine-pipeline-window throughput tracks that decomposition closely: composite reaches $7.18 \text{ TFLOP s}^{-1}$ (post-overlap limit, only 10% below the 8 TFLOP s^{-1} per-PE GEMM peak), async-full $3.91 \text{ TFLOP s}^{-1}$ (DMA and compute serialize on a single big dot), and both async-tiled variants $\sim 2.53 \text{ TFLOP s}^{-1}$ (192 commands’ worth of structural dispatch cost — even at the light per-command rate — accumulates on the wall). The next paragraph attributes those gaps to specific simulator mechanisms.

Decomposing the gap. Three structural mechanisms separate composite from the kernel-level baselines, and they layer.

1. *Inter-engine token routing happens below the*

host-side dispatch path. The composite encodes the full `DMA_READ`→`FETCH`→`GEMM`→`STORE`→`DMA_WRITE` pipeline once. The scheduler’s tile-feeder loop then emits one *tile token* per HW tile inside that one composite, and each token self-routes between engines after each stage finishes. The per-tile token routing is a scheduler-internal event, not a fresh host command, so it does not pay the structural CPU dispatch cost. At $K=3072$ the composite emits 48 tile tokens that flow fully-pipelined through five stages each — 240 inter-engine hand-offs total — behind a single command from the host’s point of view.

2. *`tl.dot` cannot replicate that per-tile pipeline at the kernel level.* A single-op GEMM command is handled on the GEMM engine as a single monolithic compute timeout for the supplied $M \times K \times N$; there is no internal token loop that would let a streaming DMA of $B[i+1]$ overlap with the GEMM of $B[i]$ inside one `tl.dot`. The user can only recover inter-tile overlap by emitting one `tl.dot` per chunk — which is exactly what the async-tiled baseline does, at the price of N host-side commands.

3. *The host-side dispatch cost the async-tiled baseline pays is structural, not modelling slack.* KernBench charges every host-emitted command a structural CPU dispatch cost $d_{\text{cmd}} = \text{FIXED} + b_{\text{logical}} \cdot R$ (§2.3.1). The cost model is deliberately charitable to the async kernels here: only a `CompositeCmd` pays the 40-cycle control-path `FIXED` (it alone drives a scheduler-built tile-feeder plan), while *every* single-op command — the 96 DMA descriptors, the 48 `tl.dots`, and the 47 elementwise adds alike — pays only the light 8-cycle `FIXED`, calibrated to descriptor-ring-push / single-instruction issue patterns in modern accelerators (NVIDIA Hopper TMA ~ 1 ISA cycle, a single `mma.sync` one instruction, AMD AQL packet writes ~ 5 –15 cycles). So the async-tiled baseline is *not* penalized by an inflated per-command cost on *any* of its operations. Yet it still emits 192 host commands against composite’s one, so ~ 192 light dispatches accumulate to $\sim 1.5 \mu\text{s}$ of structural PE_CPU time that composite never pays — composite hides its 48 tiles’ 240 inter-engine hand-offs as scheduler-internal events (mechanism 1). Layered on top, the dependency chain on the running accumulator serializes the 47 adds on the math engine. The net result is that the async-tiled kernel runs $\sim 2.8\times$ slower than composite at $K=3072$ despite getting the inter-chunk overlap right — down from $\sim 6.3\times$ under the earlier uniform-40 cost model, because D8 removed the per-command overcharge, but *not* closed: the residual gap is the command-count structure (one composite vs. 192 single-ops), not a modelling artifact.

4. *Prefetch depth is not the lever, command count is.* The depth-2 and depth- ∞ async-tiled kernels land within 1% of each other at every measured shape (Figure 7). This is the diagnostic against a natural objection: “surely the async-tiled kernel was just under-prefetching; deepen the queue and the DMA→GEMM overlap recovers.” Deepening the prefetch queue changes *when* DMA descriptors hit the engine but not their total number or

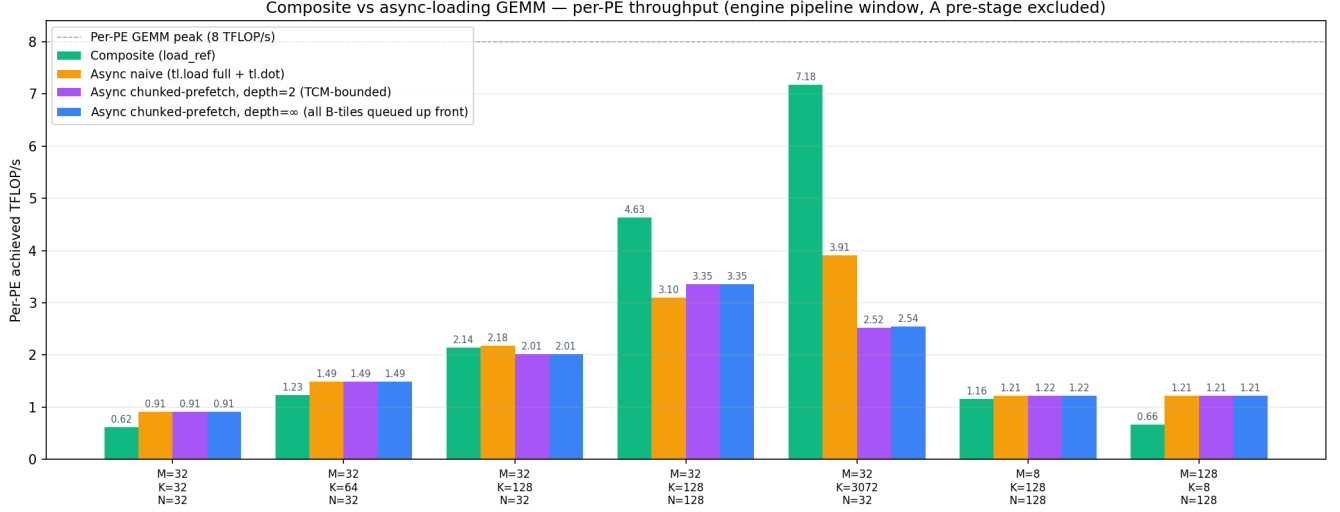


Figure 7: Per-PE achieved TFLOP/s for the same shape sweep run under four issuance patterns: composite (one command, scheduler streams per-tile internally), async-full (one `tl.dot` on fully-loaded B), async-tiled with depth-2 double-buffer (TCM-bounded; the only kernel-level variant that scales to LLM context length), and async-tiled with depth- ∞ (all B-tiles queued up front; included as a sanity check that the prefetch depth is *not* what separates the async-tiled kernel from composite). All curves exclude A ’s up-front DMA from the measurement window. Composite wins at every full-tile shape; the depth-2 and depth- ∞ async-tiled kernels deliver *indistinguishable* throughput (e.g. 2.522 vs. 2.544 TFLOP s^{-1} at $K=3072$), confirming that prefetch depth is not the lever — the structural per-command dispatch cost is. The gap between composite and the async-tiled kernels grows with K_{useful} ($\sim 2.8\times$ at $K=3072$, where the async-tiled kernels emit 192 host commands while composite emits one). The under-tile corner $M=128, K=8, N=128$ inverts: composite’s per-tile orchestration overhead exceeds the per-tile useful work, and all three async kernels beat it.

the structural dispatch cost they each pay. The $\sim 2.8\times$ gap to composite is not a prefetch-depth gap; it is the gap between “one composite command with internal per-tile token routing” and “ N_K host-side dot/add commands, each charged separately.” The depth-2 kernel additionally constrains peak TCM occupancy to $O(2 \cdot \text{TILE_K} \cdot N)$, matching composite’s per-tile streaming buffer; the depth- ∞ kernel needs the full B in TCM, which makes it infeasible at LLM context length even if the throughput were competitive.

Where the composite advantage doesn’t apply. The shape $M=128, K=8, N=128$ inverts the picture: composite delivers 0.66 TFLOP s^{-1} and both async kernels reach 1.21 TFLOP s^{-1} . The reason is consistent with the analysis above and explicit in the simulator state. Composite emits 16 tile tokens (one per output tile) for this shape, each carrying $K_{\text{useful}}=8$ MACs across a $\text{TILE_K}=64$ pipeline — 12.5% of the hardware tile’s MAC slots are useful, the rest is K-padding. The per-tile inter-engine hand-off cost stays the same regardless. When the per-tile useful work is small enough that hand-off overhead exceeds the GEMM work itself, a single-op `tl.dot` — which submits one monolithic GEMM command with no per-tile orchestration — wins. The take-away is the bound on composite’s value: it amortizes useful per-tile compute, not padding-dominated under-tile shapes. Real kernels at this corner are better served by reshape-into-batched-GEMM transforms that move under-tile K

into a tile-filling dimension before reaching the GEMM engine, which is exactly what the GQA decode kernel of §5 does for its $K_{\text{useful}} = \text{head_dim} = 128$ inner reduction.

Summary of the comparison. Composite is the only one of the four kernels to combine (a) macro-command dispatch at the host boundary (amortizing the structural CPU cost across all the work a single GEMM does), (b) scheduler-internal per-HW-tile streaming of $\text{DMA} \rightleftharpoons \text{compute}$, and (c) TCM-bounded streaming buffer. Kernel-orchestrated async kernels can have any two of those, not all three: async-full pays one host dispatch (a) but forfeits per-tile overlap (b) and pins all of B in TCM (c); depth- ∞ async-tiled achieves inter-chunk overlap but at N_K host dispatches and full- B TCM occupancy; depth-2 async-tiled fixes the TCM footprint (c) but still pays N_K host dispatches. The two corners where async catches up (small- K where composite has nothing useful to amortize; under-tile shapes where per-tile useful work is sub-token) are diagnostic of *where composite is the wrong tool*, not of a slack the user kernel could close.

4 PE_IPCQ and Collective Communication

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. Underneath the algorithm this is fundamentally a PE-to-PE problem—many short messages flowing between neighbors as the reduction proceeds. The natural software realization is a per-direction ring buffer whose head and tail pointers the producer and consumer update atomically and poll, but our H2 2025 report measured this scheme end-to-end and found that the atomic-pointer traffic together with the consumer’s polling loop dominate the per-message cost: the queue itself becomes the bottleneck well before the link runs out of bandwidth, and a pure-SW collective spends most of its time on metadata rather than on actually moving partials. A second, orthogonal problem is link sharing—if the collective rides the kernel’s generic DMA path it competes with the GEMM’s compute traffic on the same wires, so a large tile transfer head-of-line-blocks a pending reduction. This section asks how a dedicated hardware primitive can lift queue management off the software path entirely and, in the same design, stop collective traffic from stalling behind compute traffic, so the all-reduce runs overlapped with compute and at the interconnect’s physical limit.

4.1 Design

The proposed block is **PE_IPCQ** (inter-PE communication queue), a small controller dropped into every PE next to PE_DMA, PE_GEMM, PE_MATH, and PE_TCM. It is a control-plane block—it holds no payload data—and consists of three pieces: a per-direction *QPair register file* (~576 B of flip-flops covering up to eight directions), a combinational slot-address generator and backpressure comparator, and a credit injector/receiver wired to the NoC. Each QPair holds the local pointers (`my_head`, `my_tail`), shadowed views of the peer’s (`peer_head_cache`, `peer_tail_cache`), the local and peer rx-buffer physical bases, ring depth and slot size (both power-of-two), and the peer’s credit-target address. The ring data itself lives in a reserved slot region of TCM (or PE-local HBM, or cube-shared SRAM), addressed by the QPair registers; PE_IPCQ never touches the bytes, only the pointers. The PE_CPU sees the controller as an MMIO peripheral and the N/S/E/W direction labels as logical ports—one kernel image runs across a 1D ring, 2D mesh, or 2D torus because the topology only changes which peers the QPair registers point at.

Initialization. The host CCL backend brings the whole machine to a usable state before any kernel runs. `init_process_group(backend="ahbm")` loads `ccl.yaml`, resolves the algorithm + topology + buffer_kind + slot configuration, allocates an rx ring-buffer region on every participating PE so every rank now knows every other rank’s `rx_base_pa`, and fans out an `IpcqInitMsg` that writes the QPair register file on each PE_IPCQ over MMIO. The same fan-out wires the per-direction credit-return channel: each PE_IPCQ records its peer’s credit-target address and a back-pointer that

the credit receiver will use to update the right QPair. After init, every register the runtime needs is preloaded; nothing in the kernel path allocates memory, walks a table, or talks to the host.

Send. When the kernel executes `tl.send(dir="E", src_addr, nbytes)`, the PE_CPU performs a single MMIO write into PE_IPCQ describing the request (direction, source address/space, length, sender handle). The controller evaluates backpressure in one combinational compare— $(\text{my_head} - \text{peer_tail_cache}) < \text{n_slots}$ —and, on a hit, the slot-address generator returns $\text{dst} = \text{peer_rx_base_pa} + (\text{my_head} \% \text{n_slots}) \times \text{slot_size}$ in one to two cycles. PE_IPCQ then emits one `IpcqDmaToken` on its dedicated port to PE_DMA’s `vc_comm` channel, carrying the data descriptor (`src`, `dst`, `nbytes`) plus a small piggyback header (`sender_seq = my_head`, `src_coord`, `direction`). `my_head` is incremented in the same cycle—a local flip-flop bump, not a cross-PE atomic—and `tl.send` returns fire-and-forget. If the backpressure compare fails, the controller stalls the CPU in either of two modes selected at init: `poll` (CPU re-reads a status CSR) or `sleep` (controller asserts a wake event when a credit arrives). Both modes are benchmarked in the results.

Transport. PE_DMA is the only block that touches the fabric, and it has been extended for IPCQ in two ways. First, it now exposes two virtual channels—`vc_compute` for GEMM/Math `TileTokens` and `vc_comm` for `IpcqDmaTokens`—with independent state machines and a chunk-level (256 B) weighted round-robin arbiter on the shared physical link. A large GEMM tile DMA can no longer monopolize the link against a pending IPCQ send, enabling compute and communication to make progress independently as an architectural property of the design. Second, on the sender side PE_DMA packs the piggyback header into the same flit train as the data, and on the receiver side it runs the *I6 atomic terminal handler*: pay the per-flit bottleneck-BW drain, then, in one indivisible block, (i) write the payload into `MemoryStore` at `dst_addr` (the receiver’s ring slot) and (ii) forward an `IpcqMetaArrival {sender_seq, dst_addr}` to the local PE_IPCQ over a PE-internal wire. No yield, no PE_CPU interaction, no cache-coherence round trip—the bytes land in TCM and the metadata reaches the controller in the same cycle.

Receive and credit return. PE_IPCQ’s Meta Extractor range-matches the incoming `dst_addr` against each direction’s $[\text{rx_base}, \text{rx_base} + \text{n_slots} \times \text{slot_size})$ window—unambiguous even when two directions share a peer, as in a 2-rank bidirectional ring—and updates `peer_head_cache[d] := max(prev, sender_seq + 1)`, releasing any `tl.recv` blocked on direction `d`. When the kernel eventually consumes the slot, the controller increments `my_tail` and emits a 16 B credit packet on the dedicated fast-path channel preloaded at init; the latency is the real fabric path (per-node overhead + edge propagation + 16 B / bottleneck BW), so an in-cube

credit returns faster than a cross-SIP credit and the model carries no magic constants. The sender’s PE_IPCQ absorbs the credit, advances `peer_tail_cache`, and deasserts backpressure if it was stalled. The pointer-synchronization problem the software baseline solved with explicit atomic RMWs and polling loops has been split in two and dissolved into existing traffic: *head* updates ride on the DMA payload itself, with the receiver’s PE_DMA performing the data + metadata write in a single atomic step, so the sender never blocks waiting for the receiver to see it; *tail* updates ride on a dedicated 16 B credit on a side-channel, with no software in the loop. Send becomes a single MMIO write, receive becomes a flip-flop read, backpressure becomes one 64-bit subtract, and the per-message cost in IPCQ is dominated by the link traversal rather than by metadata bookkeeping—which is what the H2 2025 measurement showed the pure-software queue could not achieve. 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, with the IPCQ ring buffer placed in on-PE TCM, PE-local HBM, or cube-shared SRAM—the third knob the results section sweeps.

4.2 Design alternatives

PE_IPCQ is one point in a small space of hardware mechanisms for moving a short message from one PE to a neighbor and signalling its arrival. Three established alternatives anchor the space, each the HW realization of a familiar host-networking idea: a *doorbell + polling* scheme (the classic MMIO doorbell—write the payload by DMA, write a doorbell, let the peer poll or take an interrupt); a *hardware message queue* (HMQ, the NVLink-style descriptor engine that pushes a queue entry to the peer, with large payloads still riding a second DMA); and a *completion-queue* design (RDMA-CQ, the InfiniBand/RoCE pattern where a DMA write auto-posts a completion entry the peer’s CQ polls). PE_IPCQ is the fourth: a hardware ring with credit return, splitting the control plane into PE_IPCQ and the data plane into PE_DMA, with head updates riding the payload and tail updates riding a 16 B side-channel credit (§4).

The qualitative comparison motivates the design but is not a quantitative claim: the cycle-step tallies above are hand-counted control events, deliberately separated from the measured latencies that follow. Everything in the results subsection runs on the PE_IPCQ substrate and is simulator-grounded.

4.3 Results

All measurements in this section run on the PE_IPCQ substrate described above; the topology sweep is intended to characterize how effectively the proposed mechanism exposes the underlying interconnect’s properties, not to

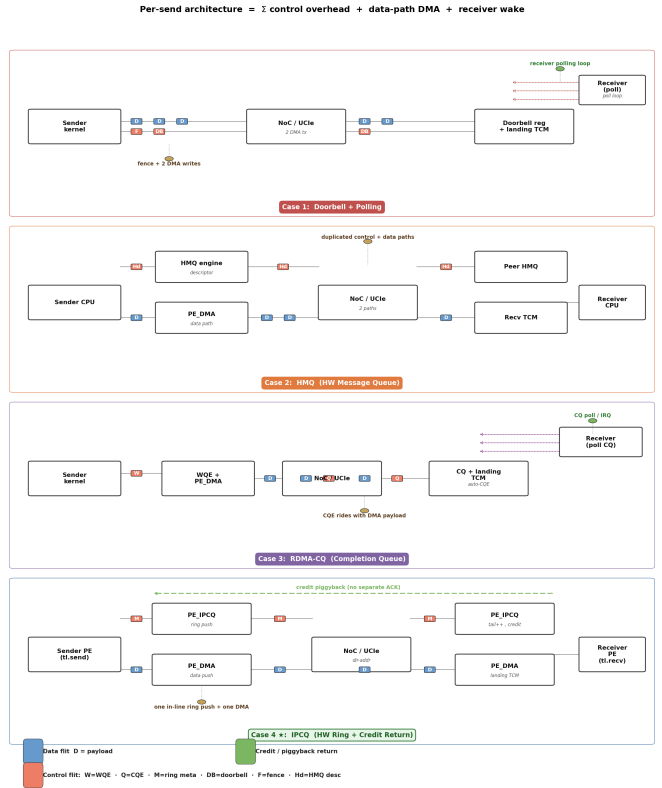


Figure 8: Per-send data and control flow for the four PE-to-PE signalling mechanisms (sender → NoC → receiver). Doorbell and RDMA-CQ each issue two fabric transactions (payload then doorbell / completion) and leave the peer polling or taking an interrupt; HMQ adds a dedicated descriptor engine but still moves large payloads on a second DMA; PE_IPCQ folds head-pointer signalling into the payload flit train and returns the tail credit on a side channel, so a send is one MMIO write and a receive is a flip-flop read. This is a *design schematic*, not a measured comparison.

compare PE_IPCQ against an alternative communication primitive. 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 2—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 (Figure 10). Table 3 and Figure 11 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 12).

Why IPCQ (HW Ring + Credit Return)? — decision matrix

Criterion	Case 1: Doorbell + Polling	Case 2: HMQ (HW Message Queue)	Case 3: RDMA/CQ (Completion Queue)	Case 4+: IPCQ (HW Ring + Credit Return)
Latency (single send)	High (2 DMAs + fence + poll/RQ)	Mid (desc relay + DMA parallel)	High (DMA + CQE + poll/RQ)	✓ Low (3 events, ~28 cy in-line)
Host CPU on critical path?	Yes (issue both DMAs + poll)	Yes (CPU builds descriptor)	Yes (post WQE + poll CQ)	✓ No (kernel-side HW push)
Polling / IRQ wake-up?	Yes (poll or RQ on doorbell)	No (CPU just reads desc ptr)	Yes (poll or RQ on CQ)	✓ No (ball advance + piggyback credit)
Duplicated control + data datapaths?	No (1 datapath, but 2 DMAs)	Yes (HMQ engine + DMA in parallel)	Partial (CQE rides with DMA)	✓ No (PE_IPCQ ctrl, PE_DMA data)
Right-sized for single-owner PE-to-PE?	Yes	Yes (but extra HW)	Multi-tenant focus (extra isolation)	✓ Yes

Figure 9: Why the ring+credit design was chosen, across five criteria: single-send latency, whether the host CPU sits on the critical path, whether the receiver must poll or take a wake-up interrupt, whether the control and data datapaths are duplicated, and whether the mechanism is right-sized for single-owner PE-to-PE traffic (rather than a multi-tenant fabric). PE_IPCQ is the only design that clears every criterion. The accompanying per-send step-count tally (~28 control events for IPCQ versus ~38 for HMQ, ~53 for RDMA-CQ, and ~56 for doorbell+polling) is an *illustrative* order-of-magnitude comparator over hand-counted pipeline steps—not a simulator measurement. The measured, simulator-grounded results follow in the next subsection.

Table 3: 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	4189	3883	2957
4,096	5566	5240	4031
16,384	10016	9376	7396
65,536	27821	25925	20858
98,304	39690	36957	29833

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.

Allreduce topology — device-level (top: ring, torus, mesh) and cube-level reduction in SIP 0

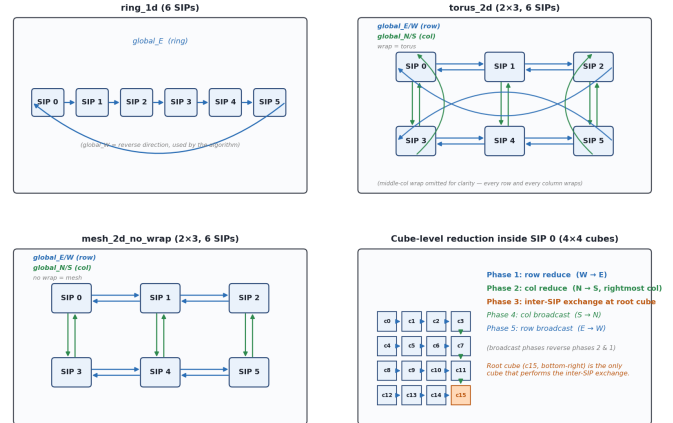


Figure 10: The three six-device (2×3) inter-device topologies the collective sweep runs over, and the hierarchical local-reduce / global all-reduce-broadcast schedule mapped onto each: a 1D ring, a 2D mesh (no wrap-around), and a 2D torus (wrap-around links on both axes). The torus’s wrap links shorten the worst-case reduction path, which is what the latency sweep below rewards.

5 Fused Grouped-Query Attention

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 $Q \cdot K^\top$ and $P \cdot V$ products, and an efficient on-device reduction (§4) for the multi-user and sequence-parallel KV reductions. *Fused* here is meant in the FlashAttention sense— $Q \cdot K^\top$, the online softmax, and $P \cdot V$ collapse into a single kernel that never materializes the score matrix—and, beyond that, the cross-device KV reduction is absorbed into the same kernel (on PE_IPCQ) rather than issued as a separate all-reduce. 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.

PE_IPCQ multidevice allreduce (Ring, Mesh, 2DTorus) vs H2 2025 single-device SW-queue baseline

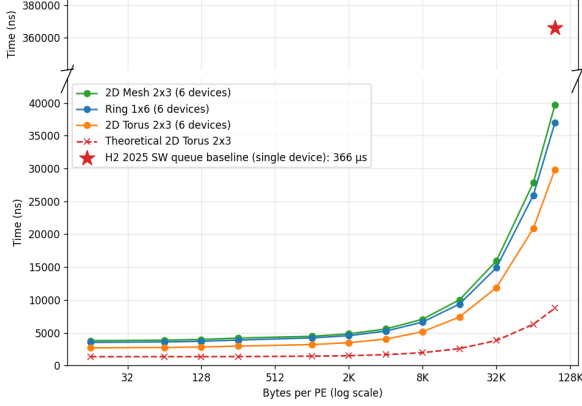


Figure 11: All-reduce latency vs. per-PE payload for the three PE_IPCQ topologies, against the analytic torus model. The isolated point in the top panel (366 μ s) is the only data point available from the H2 2025 software-queue measurement campaign—an intra-device all-reduce across 16 CUBEs on a single device. A true 6-device inter-SIP measurement under the same software queue was not collected, so this point in fact *understates* the SW-queue cost for the multidevice configuration KernBench measures here; even so the proposed PE_IPCQ inter-device curves sit roughly an order of magnitude below it, which is the headline SW-vs-HW comparison this section makes. The measured torus tracks the analytic curve within a small constant factor, the gap reflecting real link serialization that the analytic model idealizes away.

5.1 Data Placement Policy

Long-context decode is bound by the KV cache, so the first-order design question is how to place that cache—and the running softmax state it feeds—across the two hardware axes the machine exposes: the CUBEs and, within each CUBE, the PEs (here $C=8$ CUBEs $\times P=8$ PEs, for $C \cdot P=64$ attention engines over one KV-head group on the LLaMA-3.1-70B target). Each axis can *replicate* the KV cache or *shard* it, and a shard can run along the sequence dimension S_{kv} or the head dimension d_{head} . The cross product is a small, enumerable taxonomy; six placements span its meaningful corners (Figure 13).

Two quantities decide which placement is viable, and they pull against each other. The first is **per-PE KV memory**. With a per-PE HBM budget of 6.0 GB and 1.76 GB of attention weights resident, the KV headroom is 4.24 GB per PE. At a production context of $S_{kv}=1$ M tokens the unsharded cache is 40 GB/PE (Case 1), an 8-way shard is 5 GB (Cases 2–3)—both *over* the headroom—while only the 64-way placements bring it to 640 MB/PE (Cases 4–6), comfortably inside budget. Memory alone therefore eliminates Cases 1–3 at long context. The second quantity is **communication per token**, and it is what separates the three survivors. Sharding d_{head} (Cases 4–5) makes each PE hold only a slice of every head, so the QK^T score is *partial* and must be all-reduced across

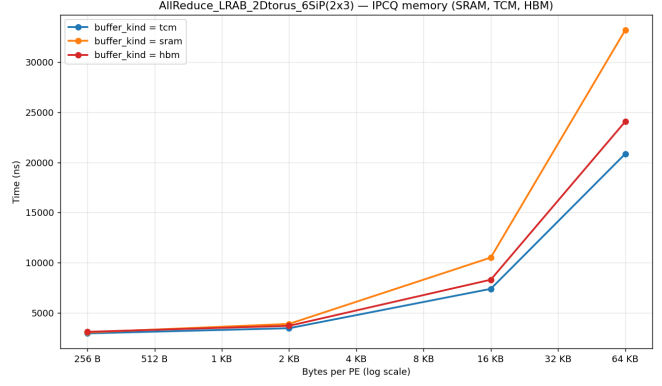


Figure 12: Effect of IPCQ staging-buffer placement (2D torus). At 64 KiB/PE, TCM staging (20 858 ns) beats HBM (24 074 ns) by $\sim 13\%$ and SRAM (33 194 ns) by $\sim 37\%$; at small payloads the three are indistinguishable.

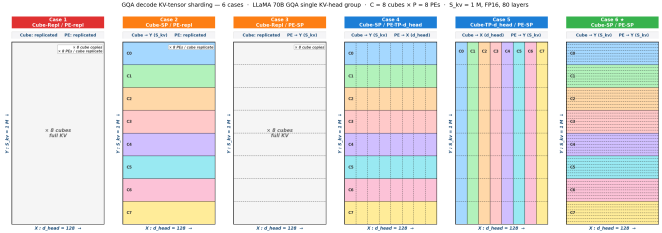


Figure 13: The six KV-placement strategies, drawn on the $S_{kv} \times d_{head}$ KV tensor (rows = sequence, columns = head dimension). Cube colour bands and dashed PE dividers show which axis each level shards. Cases 1–3 either replicate the cache or shard it on a single axis (8-way at most); Cases 4–6 reach a full 64-way split, three different ways: Case 4 splits S_{kv} across CUBEs and d_{head} across PEs, Case 5 the mirror, and Case 6 $\star$ splits S_{kv} on *both* axes.

the slice owners on every token—a reduction whose volume scales with S_{kv} (~ 166 MB/token analytically, intra-CUBE on the NoC for Case 4, inter-CUBE on UCIE for Case 5). Case 6 $\star$ instead shards S_{kv} on both axes, so every PE computes a *complete* score over its own token range and only the small running softmax state (m, ℓ, O) is merged across PEs (~ 6.2 MB/token)—a $\sim 27\times$ lighter collective than the d_{head} -split designs.

These two costs—per-PE KV memory and per-token communication—are intrinsic properties of each placement, fixed by how it shards the cache and independent of the workload regime. They do not by themselves name a winner: a short prompt where the whole cache fits on one PE values low communication and tolerates replication, whereas a million-token decode is bound by the memory wall and will pay communication to escape it. Which placement is appropriate is therefore a per-regime question, which the short- and long-context subsections that follow answer by running the options on the simulator and reading off latency, traffic, and the redundant compute each one induces.

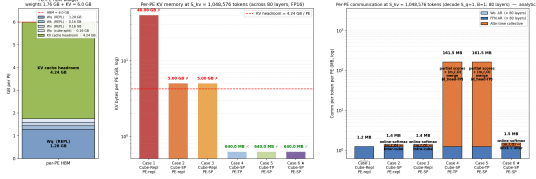


Figure 14: Long-context placement analysis at $S_{kv}=1$ M tokens. *Left:* the per-PE HBM budget—1.76 GB of attention weights leave 4.24GB of KV headroom (red line). *Middle:* per-PE KV memory per case (log scale); only the 64-way placements (Cases 4–6, 640 MB) clear the headroom, while the unsharded (40 GB) and 8-way (5 GB) cases overflow. *Right:* analytical communication per token (log scale); the $d_{\text{head-split}}$ Cases 4–5 pay a partial-score all-reduce (~ 166 MB/token) that the both-axes- S_{kv} split of Case 6 $\star$ avoids (~ 6.2 MB/token, merging only the softmax state). On these two axes Case 6 (marked $\star$ in the figure) is the single placement that lands both inside the memory budget and at low per-token communication; whether that combination is the right one to pick is a regime-specific question taken up next.

5.2 Inference with Short-Context Length

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 Inference with Long-Context Length

Long-context decode is the regime where the KV cache, not attention compute, sets serving cost, so the placement question of §5.1 becomes decisive here. To pick the right placement for this regime we run each of the six options as a fused decode kernel on the simulator (one decode step on the LLaMA-3.1-70B single-KV-head-group target, $C=8$ CUBEs $\times P=8$ PEs) at a tractable $S_{kv}=8192$, and read off end-to-end latency, on-device op traffic, and the redundant compute each one induces. The swept context is small enough that all six fit in memory at $S_{kv}=8192$;

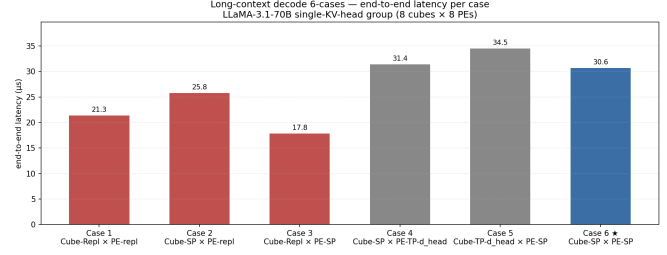


Figure 15: Measured end-to-end decode latency per placement ($S_{kv}=8192$; red = ruled out by the long-context memory budget, blue = the predicted Pareto choice). The fastest raw latency belongs to Case 3 ($17.8\mu\text{s}$)—but Case 3 replicates the full KV cache into every CUBE, so it overflows the per-PE budget at production context and wastes $8\times$ the compute (Figure 16). Among the placements that actually fit 1 M-token memory (Cases 4–6), Case 6 $\star$ is the fastest ($30.6\mu\text{s}$, versus 31.4 and $34.5\mu\text{s}$ for the $d_{\text{head-split}}$ Cases 4 and 5)—making it the placement of choice for long-context decode.

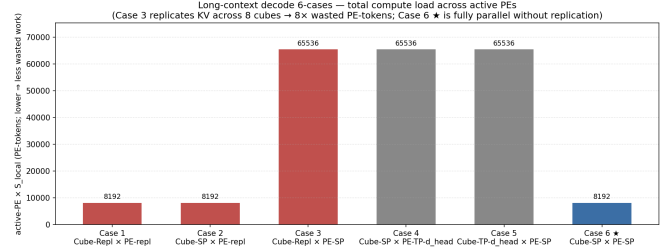


Figure 16: Redundant compute per placement, measured as active-PE $\times S_{\text{local}}$ (PE-tokens; lower means less wasted work). The minimum is 8192 PE-tokens—one pass over the sequence. Case 3 inflates this $8\times$ to 65 536 by replicating the KV cache across all eight CUBEs so every CUBE redundantly re-attends the whole sequence; the $d_{\text{head-split}}$ Cases 4–5 likewise carry 65 536 because each token is processed across eight head slices. Case 6 $\star$ achieves the full 64-way split at the minimal 8192 PE-tokens—fully parallel, no replication.

the placements the long-context memory budget rules out (Cases 1–3) are drawn in red, run here only to expose their issue and communication structure.

The measurements select the right placement for this regime. The fastest raw latency (Case 3, $17.8\mu\text{s}$) comes from replicating the full KV cache into every CUBE—which is exactly the placement the long-context memory budget forbids, and which the parallelism panel shows wastes $8\times$ the compute. Restricting attention to the placements that fit production-context memory (the 64-way splits, Cases 4–6), the choice is Case 6 $\star$: it is the fastest of the three ($30.6\mu\text{s}$), and the op-count panel shows why—its softmax-state-only reduction charges 189 IPCQ copies and one DMA write against the 280 copies and 8 DMA writes the $d_{\text{head-split}}$ Cases 4–5 pay for their partial-score all-reduce. For long-context decode, then, the appropriate data placement is the both-axes sequence shard (Case 6): it is the cheapest-communicating member of the only memory-feasible family, and the cross-

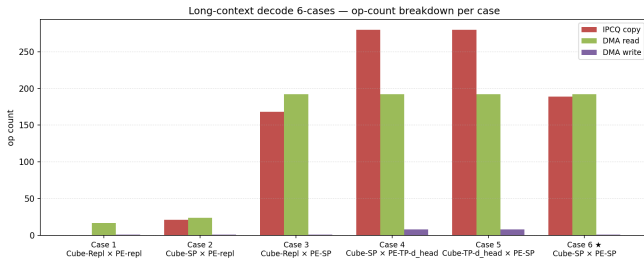


Figure 17: Measured on-device op traffic per placement. The unsharded Case 1 issues no IPCQ copies (each PE has the full cache, nothing to reduce); the single-axis Case 3 charges 168. Among the 64-way splits, the d_{head} -split Cases 4–5 charge the most—280 IPCQ copies and 8 DMA writes each, the partial-score all-reduce that head-slicing forces—while Case 6 $\star$ needs only 189 IPCQ copies and a single DMA write, because it merges just the running softmax state (m, ℓ, O) rather than partial scores. This is the on-device collective traffic that PE_IPCQ and the torus links of §4 are provisioned to absorb at link speed.

PE softmax reduction it does pay is precisely the traffic the communication-side codesign of this report is built to move quickly.

5.4 Comprehensive Analysis

These panels are the clearest statement of the codesign 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. The implication is that future hardware investment should prioritize communication and memory-system efficiency over additional compute throughput for attention-dominated inference workloads.

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.