

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?*

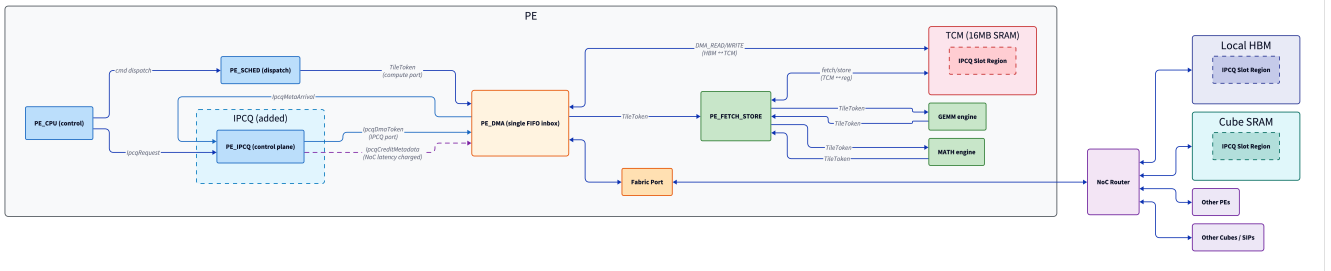
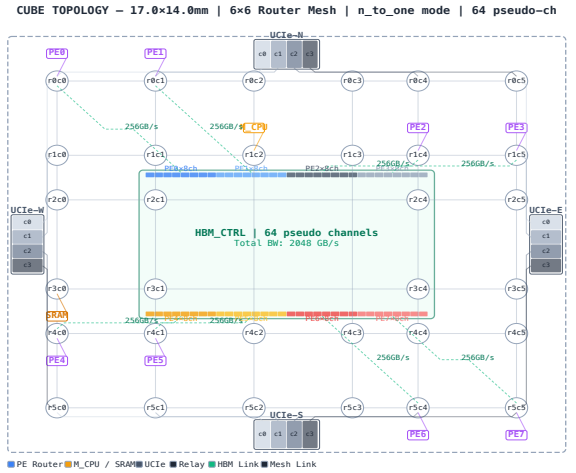
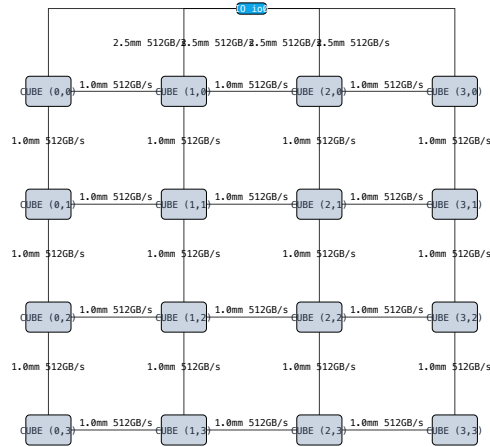
KernBench is built to answer exactly that question. Kernels are written and executed at the *source level*—as algorithmic descriptions in a small tile-oriented kernel API—with no dependency on a compiler or any other software-stack layer. The simulator takes the kernel and a hardware topology and reports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

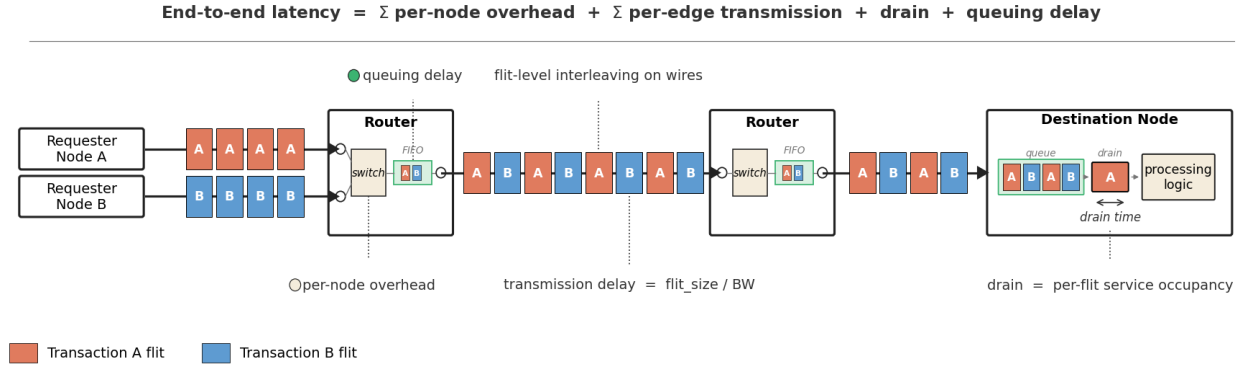


Figure 2: Conceptual schematic of the latency model. Two source nodes (Requester A/B) inject flits through a chain of routers into a destination node; on the shared edge between routers, flits from the two transactions are interleaved flit-by-flit by the wire’s FIFO arrival order. End-to-end latency 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 40-cycle **FIXED** term is the command-descriptor issue cost. 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. 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 overhead. 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 2.2 ppt across every swept shape—from a single-tile $M=K=N=32$ at $\sim 7.5\%$ up to the deep- K $K=3072$ case at $\sim 88\%$. The residual gap is fill/tail overhead the closed-form pipeline model omits.

Second, in the all-reduce study (§4, Fig. 5), 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 UCIE-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.

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 consis-

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

tent 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, $8 \text{ PEs} \times 8 \text{ TFLOP s}^{-1}$ 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 $\sim 31 \text{ FLOP/byte}$; 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.

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

Table 2: Modeled hardware configuration

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
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 command	40 cycles
per-byte rate R	$0.0625 \text{ cycles/byte}$ (16 B cycle^{-1})
composite size cap	1024 B

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?

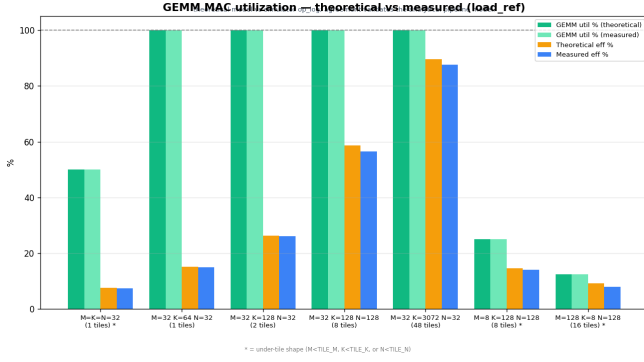


Figure 3: GEMM MAC utilization and efficiency, theoretical vs. measured (load_ref staging). Tile-fill sets the ceiling: under-tile shapes (marked *) such as $M=K=N=32$, $M=8$, and $K=8$ cannot fill the MAC tile and cap at 50 %, 25 %, 12.5 %. For tile-filling shapes, efficiency climbs with tile count—from ~15 % at one tile to ~88 % measured at 48 tiles ($K=3072$)—and the measured bars track the analytic ideal-pipeline prediction within 2.2 ppt across every shape.

3.1 Design

The answer is the *composite command*. A single command carries the ordered 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 $\text{DMA} \rightarrow \text{fetch} \rightarrow \text{GEMM} \rightarrow \text{store}$ 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).

3.2 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 3 reports MAC utilization and efficiency, and Figure 4 breaks the kernel into per-stage engine busy time.

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

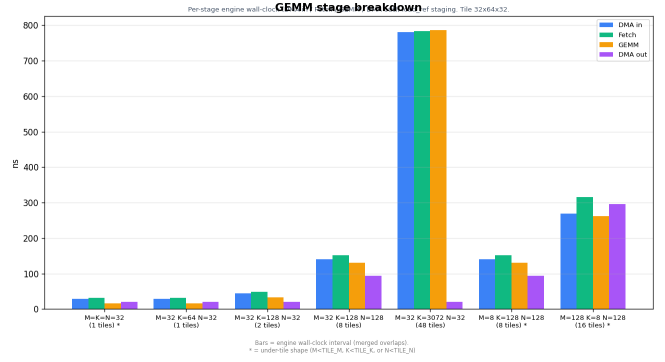


Figure 4: Per-stage engine wall-clock (DMA in, Fetch, GEMM, DMA out) under load_ref staging. For the deep- K shape ($K=3072$, 48 tiles) DMA in, Fetch, and GEMM are all close to ~785 ns, running concurrently in a tightly pipelined regime while DMA-out is negligible. For the low-reuse shape ($M=128, K=8, N=128$, 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.

tiles amortize the one-time pipeline fill, so the deep- K shape reaches ~88 % of peak while a single-tile shape reaches only ~15 %. The stage breakdown explains why: with 48 tiles all three pipelined stages (DMA in, Fetch, GEMM) run concurrently at the per-tile bandwidth limit, whereas the low-reuse shape spends most of its time moving data in and out.

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 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 (~88 % 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 3) 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 PE_IPCQ and Collective Communication

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.

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 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. Table 3 and Figure 5 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.

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

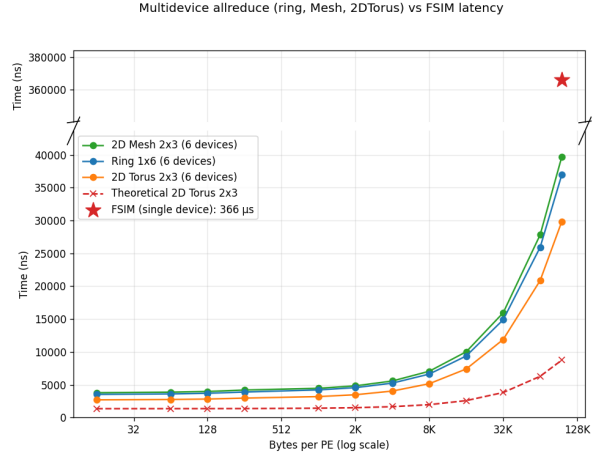


Figure 5: All-reduce latency vs. per-PE payload for the three topologies, against the analytic torus model and an external full-system simulator (FSIM) reference. The measured torus tracks the analytic curve within a small constant factor; the FSIM single-device point (366 μ s) sits an order of magnitude above the KernBench algorithmic latency, illustrating the difference between an achievable-kernel number and a full end-to-end-stack number.

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 6).

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.

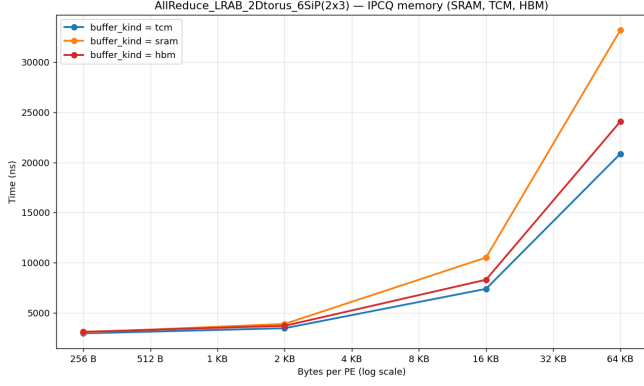


Figure 6: Effect of IPCQ staging-buffer placement (2D torus). At 64 KiB/PE, TCM staging (20858 ns) beats HBM (24074 ns) by ~13% and SRAM (33194 ns) by ~37%; at small payloads the three are indistinguishable.

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 (§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

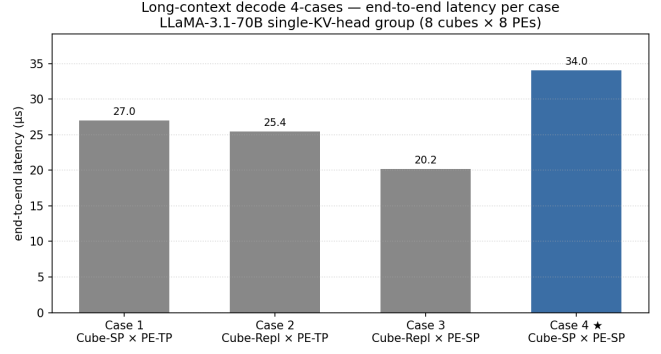


Figure 7: End-to-end decode latency per parallelism strategy (LLaMA-3.1-70B single-KV-head group, 8 CUBEs × 8 PEs). Replication into CUBEs (Cases 2/3) wins the latency race (20.2 μs for Case 3), but Case 4 (*, KV split 64-way) finishes within 14 μs of the leader while paying a different cost—visible in Figure 8.

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: long-context decode and parallelism strategies

The four headline panels above stress the kernel at moderate context lengths. Long-context decode—the regime where KV cache size, not attention compute, sets serving cost—turns the choice of how to parallelize across cubes and PEs into a first-order design knob. We compare four strategies on the LLaMA-3.1-70B single-KV-head-group target (8 CUBEs × 8 PEs, one KV-head group):

- **Case 1** (Cube-SP × PE-TP): KV split by S_{kv} across CUBEs; PEs tensor-parallel on the batch dimension (wastes PE-TP work at $B=1$).
- **Case 2** (Cube-Repl × PE-TP): full KV replicated to every CUBE; PEs tensor-parallel on batch.
- **Case 3** (Cube-Repl × PE-SP): full KV replicated; PEs sequence-parallel on S_{kv} with an intra-CUBE all-reduce.
- **Case 4** (*, Cube-SP × PE-SP): KV split 64-way (across both CUBEs and PEs) with a two-phase all-reduce on the running softmax state (m, ℓ, O).

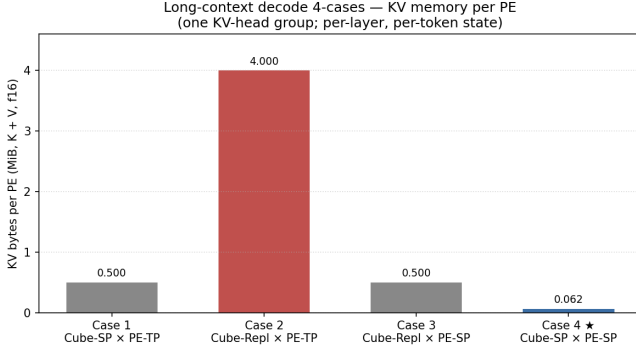


Figure 8: Per-CUBE KV memory footprint for the four cases. Cases 1 and 4—both with the KV cache split across cubes (Cube-SP)—hold only 0.5 MiB of KV state per CUBE; Cases 2 and 3, which replicate the full KV, hold 4 MiB per CUBE, an $8\times$ blowup at this configuration that scales linearly with context length.

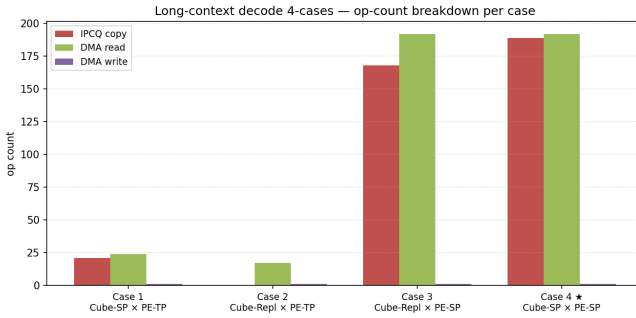


Figure 9: Per-case op-count breakdown. The replicated-KV PE-TP design (Case 2) avoids almost all on-device communication (~ 0 IPCQ copies), but at the cost of KV memory. Case 4’s two-phase reduce charges ~ 190 IPCQ copies and ~ 190 DMA reads—this is the traffic that PE_IPCQ (§4) is built to absorb at on-device speed.

Three things stand out. First, the fastest case in pure latency (Case 3, $20.2\mu\text{s}$) is also the most memory-hungry, requiring the full KV state on every CUBE—an option that fails to scale once context length blows past the per-CUBE budget. Second, Case 4’s KV-split design gives back roughly $14\mu\text{s}$ versus Case 3 in exchange for an $8\times$ KV-memory reduction; for practical long-context serving where KV capacity is the binding constraint, this is the trade the design chooses (marked $\star$). Third, Case 4 pays its way in *communication*: the op-count panel shows ~ 190 IPCQ copies and ~ 190 DMA reads, precisely the on-device collective traffic that PE_IPCQ and the torus links of §4 are provisioned to move quickly—so the “slower” strategy is in fact the one that fully cashes in the communication-side codesign work of this report.

5.4 Analysis and meaning

These panels are the clearest statement of the codesign thesis in the report. Because the composite com-

mand 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 $\sim 78\%$ 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.