PDiMC: Achieving High-Throughput LLM Inference and Resolving Memory Concurrency via an Efficient PIM Subsystem

Byeongho Kim
Samsung Electronics
South Korea
otribute@gmail.com

Sukhan Lee
Samsung Electronics
South Korea
infinity1026@gmail.com

Wontak Han
Samsung Electronics
South Korea
wtak.han@samsung.com

Hweesoo Kim
Samsung Electronics
South Korea
hs5006.kim@gmail.com

Yoonah Paik
Samsung Electronics
South Korea
hmhmhey@gmail.com

Jaewan Choi
Samsung Electronics
South Korea
jw10.choii@gmail.com

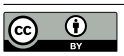
Kyomin Sohn
Samsung Electronics
South Korea
kyomin.sohn@samsung.com

Abstract

Large Language Model (LLM) inference is latency-sensitive and often memory-bound due to frequent accesses for matrix-vector multiplication operations. Processing-in-Memory (PIM) offers a promising solution by leveraging internal bank parallelism. However, its practical adoption in general-purpose systems is severely constrained by strict instruction ordering and critical interference from concurrent memory workloads, which can disrupt PIM command sequences and compromise system stability.

To address these challenges, we propose PDiMC, a novel PIM-DMA integrated Memory Controller. Unlike previous approaches that rely on conventional memory controllers (MCs) and costly software synchronization, PDiMC introduces a lightweight controller that directly issues DRAM commands by sharing the PHY interface via a DFI MUX. This architecture bypasses the conventional MC's scheduling to inherently guarantee strict command ordering and effectively isolate PIM traffic from normal commands. Furthermore, PDiMC employs a tunable interleaving mechanism to enable the safe and concurrent execution of PIM and standard memory operations.

Fully compatible with JEDEC standards, PDiMC achieves near-ideal GEMV acceleration, delivering up to 68% speedup over software-controlled PIM approaches and falling only 4.4% short of a fence-free (functionally incorrect) execution. In multi-programmed environments, PDiMC improves LLM inference throughput by up to 2.8× over a non-PIM baseline, while increasing the IPC of co-running applications by up to 13.5%. PDiMC offers a practical approach to unlocking PIM's potential without compromising system performance or stability.



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846842

CCS Concepts

• **Hardware** → **Emerging architectures**; • **Computer systems organization** → *Embedded software*; • **Computing methodologies** → *Neural networks.*

Keywords

Processing-in-memory (PIM), On-device LLM inference, Memory Controller, Direct Memory Access

1 Introduction

Recently, AI has seen a surge in popularity with the prominence of Large Language Models (LLMs) such as GPT [43], Gemini [59], Claude [1], DeepSeek [6], and Llama [40]. As the number of parameters increases steadily, these models demand intensive computation and memory resources, particularly during the decode phase [55]. In this phase, LLMs generate output tokens iteratively, a process dominated by General Matrix-Vector Multiplication (GEMV) operations in both Feed-Forward Networks (FFNs) and Attention layers as shown in Figure 1. Unlike the prefill phase, where larger numbers of tokens may benefit from General Matrix-Matrix Multiplication (GEMM), the decode phase operates with a single token, especially when responding to user input in real time [5]. This single token, or single batch, processing pattern is particularly prevalent in on-device AI environments, where maintaining low latency and ensuring quality of service (QoS) are critical [42]. As a result, this phase exhibits memory-bound behavior, where the overhead of frequent memory accesses outweighs the compute costs [48].

Processing-in-Memory (PIM) has emerged as a promising solution to these challenges by moving computation closer to memory and exploiting high internal bandwidth. GEMV operations, a core bottleneck in LLM inference due to limited weight reuse and high memory access frequency [46], are particularly well suited for PIM acceleration. Among various PIM architectures, those that place processing units physically near DRAM banks have demonstrated practical viability through real hardware prototypes and commercial deployments. By offloading memory-bound operations

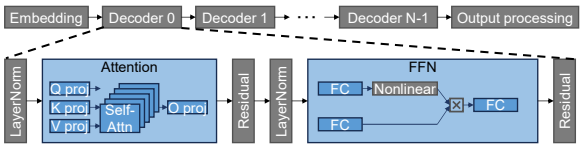


Figure 1: Structure of Transformer-based LLM. Blue-colored layers are mainly composed of GEMV operation.

to PIM, significant improvements in throughput and latency can be achieved, even outperforming idealized compute-bound processors limited by external memory bandwidth. This makes PIM a compelling complement to conventional accelerators such as NPUs and GPUs, especially in heterogeneous systems where workload characteristics can be dynamically mapped to the most suitable computing unit. As LLMs continue to grow in scale and complexity, the synergy between their computational patterns and the architectural strengths of PIM presents a timely opportunity for next-generation AI acceleration [55].

Numerous prior works have demonstrated the potential of PIM to accelerate a variety of applications [12, 26, 37, 47, 53, 65, 66]. In addition, several studies have proposed mechanisms to support the coexistence of PIM-related memory requests and conventional memory accesses within the same system, aiming to enable seamless integration of PIM into heterogeneous computing environments [15, 52, 63, 64]. However, many of these approaches introduce additional hardware features or modify the DRAM interface beyond the JEDEC standard, such as custom signaling protocols or extended instruction sets. As a result, these designs often sacrifice compatibility with commodity DRAM systems, making them difficult to integrate into existing computing platforms without significant architectural changes. In contrast, PIM architectures such as HBM-PIM [33] are specifically engineered to function as drop-in replacements for standard HBM modules. This design philosophy aims to minimize costly host-side hardware modifications, thereby facilitating the demonstration of performance and energy efficiency gains achievable through PIM adoption. These architectures support both standard memory operations and PIM execution by toggling between normal and PIM modes. This dual-mode capability ensures backward compatibility with conventional memory controllers (MC) and software stacks.

However, the practical deployment of these dual-mode PIM architectures in general-purpose, multi-programmed systems faces critical challenges regarding both performance and system stability. First, to ensure the strict DRAM command ordering required for PIM mode changes, developers must rely on costly software-level memory ordering primitives (memory fences), which heavily degrades the performance benefits of PIM. Second, and more critically, while prior studies have proposed lightweight memory ordering primitives [41], they still rely on the conventional memory controller (MC). In modern OS-driven environments, random memory accesses from concurrent, non-AI processes share the same memory channel. When these normal requests interleave with the carefully orchestrated PIM command sequences inside the conventional MC's scheduling queue, they cause catastrophic functional errors that would inevitably manifest as incorrect OS behavior or severe system crashes in real-world deployments. Without ensuring strict hardware-level isolation, PIM cannot be safely adopted as main memory in multi-tenant systems.

Table 1: Comparison of Industry PIM Architectures

Feature	**S-PIM** [33]	**H-PIM** [34]	**U-PIM** [7]
Interface	JEDEC-Compatible	Extended command bit	JEDEC-Compatible
Execution Model	Host-controlled	Host-controlled	Independent DPU execution
Memory Type	HBM2	GDDR6	DDR4
Compatibility	Standard HBM2	Requires MC modification	Standard DDR4 RDIMM

To address these fundamental limitations, we propose **P**IM-**D**MA **i**ntegrated **M**emory **C**ontroller (PDiMC), a novel system architecture that maximizes PIM benefits while guaranteeing system stability in concurrent environments. Rather than routing PIM requests through the conventional MC, PDiMCfeatures a lightweight, PIM-specific controller that issues optimized DRAM command sequences directly to the PHY interface using a DFI MUX. This architectural bypass natively enforces strict command ordering without software fences and completely isolates PIM traffic from conventional memory requests. Furthermore, to prevent CPU starvation during prolonged PIM execution, PDiMC incorporates a tunable interleaving scheduling mechanism that safely time-multiplexes PIM and normal DRAM requests, guaranteeing Quality of Service (QoS) for co-running applications.

This paper presents PDiMC, addressing the need for efficient PIM utilization. Our key contributions include:

- We demonstrate that the reliance of existing PIM solutions on conventional MCs not only incurs severe mode-switching overheads but also leads to critical system-level faults due to command interleaving in multi-programmed environments.
- We propose PDiMC, a JEDEC-compatible architecture that utilizes a DFI MUX to bypass the conventional MC. This design directly issues DRAM commands for PIM execution, thereby eliminating software fence overheads and intrinsically isolating PIM operations from host memory traffic.
- We introduce an interleaved PIM scheduling mechanism that dynamically balances PIM and normal DRAM requests, effectively supporting true concurrent processing while guaranteeing QoS for standard workloads.

2 Background

2.1 Real-world PIM Implementations

Various implementations of PIM architectures have been proposed as summarized in Table 1, particularly in industry, with notable examples including Samsung's HBM-PIM [33], SK Hynix's GDDR6-AiM [34], and UPMEM's DDR-based PIM solution [7]. While these designs all incorporate processing units within or near DRAM arrays to enable in-situ computation, they differ significantly in terms of processing unit integration, host interface, and execution model. A common structural feature among these designs is the placement

of processing units adjacent to memory banks, allowing each bank to transfer data locally to its associated compute unit and perform operations in a bank-parallel fashion. For instance, in GEMV, a representative memory-bound operation [5], the matrix data is distributed across banks, while the vector is stored in either global or per-bank registers. The PIM units fetch data from these sources, carry out computation, and write the results back to memory, from which the host retrieves the final outputs.

Samsung's HBM-PIM [33] (S-PIM) supports the JEDEC standard interface [18], allowing backward compatibility with conventional MCs and enabling drop-in replacement of standard HBM2 modules. PIM operations are triggered via standard memory commands (ACT, PRE, RD, WR), and require only driver-level modifications to assign address regions for PIM. A key feature of this approach is the JEDEC-Compatible PIM Protocol (JCPP), which uses special address mappings to distinguish PIM operations from normal memory accesses. Mode switching between normal and PIM mode is performed via dedicated row and bank activations, allowing memory and PIM operations to coexist without requiring changes to the host MC.

Diverging from strategies prioritizing drop-in compatibility, SK Hynix's GDDR6-AiM [34] (H-PIM) embeds PIM capabilities by extending the foundational GDDR6 [20] command protocol. By leveraging reserved command bits to introduce dedicated PIM instructions tailored for deep learning workloads, the H-PIM architecture circumvents the need for address-based mode switching or MMIO (Memory Mapped I/O)-style control interfaces. This design empowers the MC to directly orchestrate PIM computations and associated data transfers using explicit commands. While this approach yields advantages in lower operational latency and more granular control compared to architectures reliant on mode-switching semantics, it necessitates substantial modifications to the MC, introducing new operational states and protocol handling.

UPMEM proposes a JEDEC-compatible DDR [19]-based PIM architecture [7] (U-PIM) where each DRAM bank is paired with a small RISC-V core, known as a DPU (Data Processing Unit). Unlike the two prior architectures, which are tightly coupled with the host system and DRAM controller, DPUs operate as fully independent co-processors, executing offloaded code autonomously. The programming model is akin to CUDA [38]: developers compile PIM-specific kernels with U-PIM's toolchain, and these kernels are then dispatched to DPUs. Each DPU performs computations independently and signals the host upon completion. While this approach allows greater flexibility in processing, the independence of DPUs complicates the dual use of DRAM as both main memory and compute substrate, limiting the compatibility of U-PIM with conventional memory systems.

Considering the passive nature of memory systems and the need for compatibility with existing memory architectures, an S-PIM-like JEDEC-compatible PIM architecture is regarded as the most viable approach. Therefore, as shown in Figure 2, we adopt an S-PIM-like architecture as the baseline for our study. For the base DRAM, we assume an LPDDR5-based [21] configuration, which aligns with the directions reported by two major industry memory vendors [25, 29].

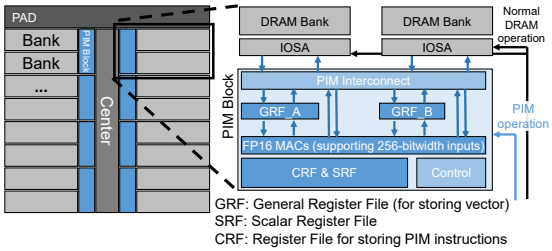


Figure 2: Concept of LPDDR5-based PIM architecture, LPPIM. Our design assumes using commodity DRAM using S-PIM [33] style PIM architecture.

2.2 PIM Architecture and Operation Flow

This section describes the hardware structure and operation flow of the base model for PIM in this work, named LPPIM (LPDDR5-based PIM architecture), which adopts a design similar to S-PIM, as illustrated in Figure 2. LPPIM integrates one PIM unit per two LPDDR5 banks, enabling concurrent operation across half of the total banks. This bank-level parallelism allows LPPIM to deliver up to 4× bandwidth compared to conventional LPDDR5. Each PIM unit is composed of a controller, a register file, and a SIMD-based arithmetic unit. The controller orchestrates PIM operations including pipeline control and data forwarding. The register file consists of three components: the Command Register File (CRF), General Register File (GRF), and Scalar Register File (SRF), which store PIM instructions, vector data (input/output), and scalar inputs, respectively. The SIMD ALU supports FP16-based element-wise operations (MULT and ADD) and Multiply-Accumulate (MAC) computations between register-stored data and memory-resident data.

From a software perspective, the address space is logically divided into the standard DRAM cell region and a PIM-dedicated region. Operations such as register read/write, configuration, and mode control are performed via accesses to the PIM-dedicated address space. Internally, this PIM-dedicated region is realized by allocating additional rows and columns within the DRAM array, outside the normal DRAM cell area (see Figure 3). When the host accesses this PIM-dedicated region, the MC issues row activate (ACT), row precharge (PRE), read (RD), and write (WR) commands to the designated additional rows/columns. Given that JEDEC specifications allow more row bits than typically used for chip capacity, this address mapping strategy using otherwise unallocated address space is feasible and does not interfere with normal DRAM operation.

Figure 4 shows the operation flow of LPPIM for accelerating GEMV, its main acceleration target. The operation begins with precharging all DRAM banks, aligning all banks at the same state and getting ready for mode change.

The PIM preparation stage is accomplished in HAB mode, which is entered by activating designated rows on the specific banks. In HAB mode, the host configures the register files in the PIM unit. This configuration includes loading scalar/vector inputs and instructions into SRF, GRF, and CRF. The write sequence to these registers in PIM units is transmitted to all banks in parallel. Writing a value of 1 to a specific control register (PIM_OP_MODE) transitions the

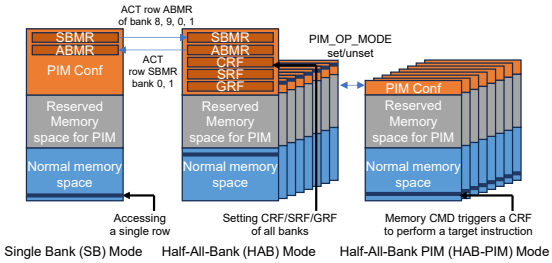


Figure 3: Memory address space and PIM-related modes of LPPIM, which is inherited from S-PIM. Blue colored arrows show how the host can alter LPPIM's mode among single-bank (SB), multi-bank (HAB), and HAB-PIM mode.

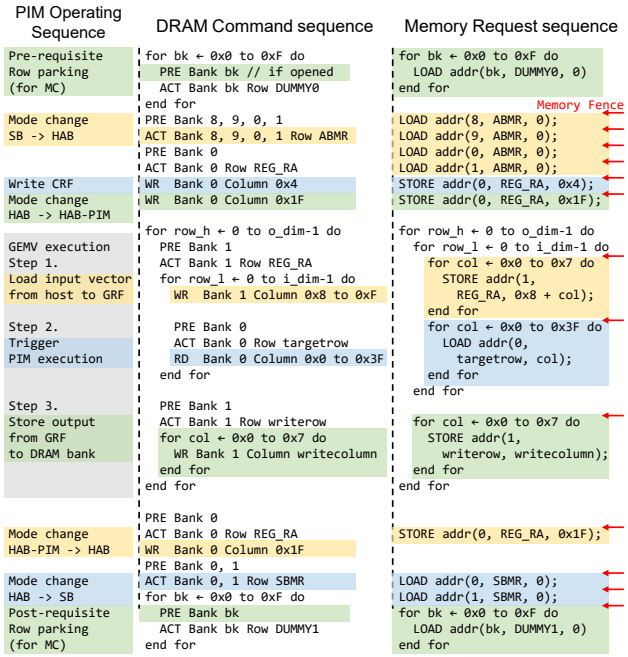


Figure 4: "Rosetta Stone" for LPPIM's operation flow, corresponding DRAM command sequence, and host-side load/store sequence for processing GEMV. Sequences located horizontally with the same color indicate the same operation. Red-colored arrows point to where a fence operation is required for ensuring DRAM command ordering. Unnecessary DRAM commands are omitted for simplicity.

PIM into HAB-PIM mode, enabling simultaneous data movement between the DRAM cell and the PIM unit for computation.

Once the system enters HAB-PIM mode, GEMV execution begins as shown in the gray box of Figure 4. In *Step 1*, the MC issues WR commands to load a portion of the input vector into the In GRF. *Step 2*, the MC sends RD commands, each of which triggers the execution of a corresponding instruction in the CRF, initiating a MAC operation between the input vectors in the GRF/SRF and the matrix weights stored in the DRAM cells. These computations are carried out within the SIMD ALU of each PIM unit. Steps 1 and 2 are

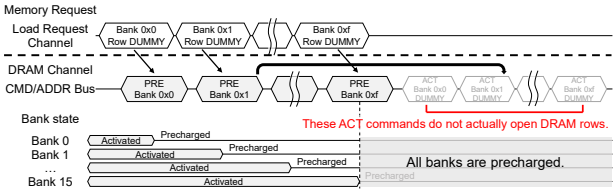


Figure 5: Memory request and command sequences for "row parking". After this procedure, all banks are in a precharged (closed) state.

repeated in proportion to the size of the input vector (i_dim). Once a portion of the output vector has been fully computed and stored in the GRF, *Step 3* is performed: the MC issues WR commands to spill the output vector back into the corresponding DRAM odd bank, row (writerow), and column (writecolumn). The full GEMV execution consists of repeated iterations of Steps 1-3, with the number of iterations proportional to the size of the output vector (o_dim). After all outputs have been written back to memory, the system exits HAB-PIM mode and reverts to SB mode. After that, the host reads the finalized output vectors to complete the GEMV process.

This execution model, in which a single input vector is broadcast to all PIM units with weights resident in the banks, targets GEMV. So the single-batch, memory-bound decode we evaluate is the regime where PIM is most advantageous. Operations with higher arithmetic intensity, such as the GEMM of the prefill phase or of batched decoding, amortize weight fetches across multiple tokens and are therefore better served by a compute-oriented accelerator; commodity PIM devices, including S-PIM and LPPIM, do not accelerate such operations. Our work maximizes what such a device can do under a general-purpose host rather than extending the acceleration envelope of the device, so batched and prefill execution are outside the scope of this study.

2.3 Methodology for Controlling PIM

Modern computing systems organize their memory subsystems as hierarchical structures that include fast, SRAM-based caches and larger external DRAM, connected to the host processor through a dedicated memory channel [17]. At the center of this memory subsystem is the MC, which does much more than simply forward read and write commands. Beyond basic command handling, the MC performs essential tasks such as translating physical addresses into DRAM-specific formats, scheduling commands with precise timing constraints, managing queues for memory access, and issuing regular refresh commands to preserve data integrity, as shown in Figure 6. These tasks are handled by internal components like arbiters, schedulers, request FIFO queues, and command generators, which deliver DRAM instructions to the PHY interface using the standard DFI protocol [44]. The PHY then converts these commands into electrical signals that comply with JEDEC standards.

LPPIM follows JCPP and is designed as a drop-in replacement for LPDDR5, enabling LPPIM to operate with standard memory controllers without requiring modifications. Control of PIM must be based on DRAM command sequences, which are driven by the load/store request sequences issued by the host, as illustrated in

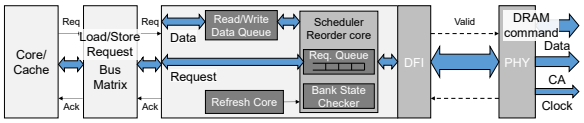


Figure 6: A block diagram of general DRAM controller

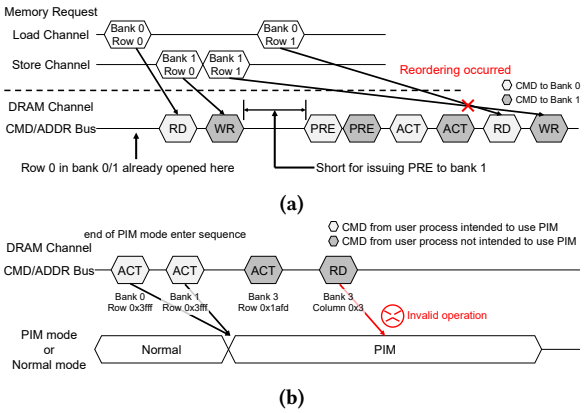


Figure 7: Limitations of using PIM under conventional memory systems. (a) Reordering feature and (b) Source unawareness

Figure 4. The RD/WR commands required for PIM operation are driven by the corresponding load/store requests directed at the memory address. Conversely, because a conventional MC cannot issue ACT commands on its own, the host triggers these ACT commands by sending a load request that accesses data from the target row.

All banks should be precharged both before and after PIM execution to ensure a valid PIM mode change and to match the bank state tracked by the MC with the actual DRAM bank state. If the two states do not match, the MC may send non-PIM commands after the PIM mode change, and these commands will eventually be misinterpreted as PIM commands. Also, if the MC utilizes an adaptive open-page policy, a row typically remains open because the MC cannot issue a PRE command on its own until bank timeout. Thus, prior work [51] has proposed *row parking*, which accesses dummy addresses in the PIM-dedicated region across all banks. By doing so, the MC is tricked into perceiving that the dummy rows are open while all banks are physically precharged (See Figure 5). Because row parking involves only a small number of dummy rows, the precharge overhead it incurs is negligible.

3 Analyzing Practical Issues of PIM

3.1 Limitations in Integrating Current PIM Architecture

While PIM architectures are allowed to operate on commercial systems without custom MCs, their performance potential is often significantly undermined by conventional MC behavior. One representative MC behavior is command reordering [50], as illustrated in Figure 7(a). The timing parameters for RD/WR commands vary depending on the condition of the target address, particularly the target bank and target row. As a result, when using widely adopted policies such as First-Ready First-Come-First-Serve (FR-FCFS), a memory request issued later can be serviced before an earlier one. Similarly, reordering can affect ACT and PRE commands as well, meaning that the sequence of load/store requests may no longer guarantee the intended DRAM command sequence. For LPPIM to function correctly, DRAM commands must be issued in a predefined sequence. However, due to command reordering, this sequence may be disrupted, potentially causing unexpected behaviors in LPPIM (e.g., mode change failures due to incorrect ordering of bank activations).

To enforce the requisite command ordering and prevent MC-induced reordering hazards, developers typically resort to inserting software-level memory ordering primitives (e.g., mfence on x86). Although functionally effective in guaranteeing order, this software-based workaround, which should be added at the red-colored arrow in Figure 4, incurs substantial performance penalties due to the overhead inherent in fence instructions. We quantified this overhead through latency analysis on a GPU platform issuing memory request sequences for PIM-based GEMV operations. Comparing execution latencies with and without explicit fence insertion demonstrated an average latency penalty of approximately 100 ns per fence, corroborating findings from prior work [33, 41]. This fence overhead rapidly accumulates, establishing a significant performance bottleneck. Compared to an idealized execution without explicit ordering primitives, the latency imposed by fence instructions alone accounts for approximately 44% of the total PIM kernel execution time (for details, refer to Section 6). These profound overheads drastically curtail the practical realization of the performance advantages theoretically offered by drop-in JCPP PIM architectures under conventional memory control paradigms.

Various solutions have tried to remove this fence overhead. Orderlight [41] introduced a new lightweight memory ordering primitive designed to enforce memory-centric ordering at the MC level, rather than relying on memory fences. This imposes substantial design cost and integration complexity, as it requires support across the entire memory hierarchy, including cores, caches, and interconnects. Direct Memory Access (DMA) based solutions [27, 28, 36, 45], which enable high-throughput data transfers between memory and peripherals without CPU intervention, have also been proposed. By offloading the issue of memory requests for the PIM architecture to a DMA engine, the host-side memory system (e.g., L1/L2 cache) can be bypassed by non-temporal access, thereby mitigating the overhead associated with software-level memory fences. However, since both solutions still rely on conventional MCs, it remains necessary to enforce correct command ordering at the MC level to prevent reordering, which introduces additional overhead and complexity.

3.2 System Integration Challenges for PIM as Main Memory

As highlighted by previous studies [25, 41, 62], beyond performance concerns, using PIM as part of the main memory subsystem introduces critical challenges to system stability and compatibility, especially in general-purpose computing environments. In practice, unless a system integrates a dedicated PIM module separate from

the main DRAM, PIM and standard memory functions must share the same physical memory interface. This sharing becomes problematic under the JCPP model, where a DRAM command by itself does not indicate whether it is intended for PIM mode or normal mode; the interpretation of the command is determined solely by the mode the memory currently resides in, which in turn depends on whether a predefined mode-switching sequence has been issued beforehand.

In multi-process systems, such as those running full-scale operating systems, memory channels are often accessed concurrently by the OS and multiple IPs. During PIM execution, random DRAM accesses from unrelated processes can interrupt the carefully controlled command sequences, leading to functional errors. Such accesses are not limited to the explicit loads and stores of co-running applications. A representative example is address translation: a TLB miss triggers a hardware page-table walk that reads DRAM. These page-table-walk accesses are hardware-initiated, not software-requested. They therefore arise whenever a process is scheduled, and the process using PIM cannot suppress them.

As a concrete failure scenario, when the system enters HAB mode for PIM operations, a standard memory request to one of the banks can cause incorrect data to be returned to the MC, as shown in Figure 7(b). This incorrect data may result in system-level faults such as incorrect OS behavior or, in extreme cases, kernel panics that crash the entire system. Although both the OrderLight-style and PIM-DMA-style approaches have proven effective in alleviating performance limitations in single-process systems, they do not resolve the reliability issues observed in multi-process environments. These solutions are not designed to prevent cross-process interference that disrupts PIM command sequences, leaving the system vulnerable to faults.

To prevent such misinterpretation between PIM and non-PIM transactions, one possible workaround is to block all DRAM activity from the OS and other IPs during PIM execution. However, such blocking would require a substantial overhaul of the software stack and would compromise the SoC's general-purpose capabilities. Practical deployment of PIM as a general-purpose main memory technology therefore hinges on isolating PIM transactions from non-PIM transactions. Achieving this isolation without sacrificing the host's general-purpose nature remains a non-trivial challenge in multi-tenant and mobile environments.

4 PIM-DMA-integrated MC

For the reasons mentioned above, it is more efficient for a dedicated PIM control unit, rather than the host processor, to control PIM memory. In this section, we introduce the architecture of PDiMC, a unit that combines a lightweight memory controller and a PIM-specific command generator to efficiently manage PIM execution while remaining compatible with conventional memory systems.

4.1 Design Considerations for PDiMC

A dedicated controller for PIM can manage memory in two principal ways: (1) by issuing memory requests from a PIM-aware DMA device, similar to the host but optimized for PIM workloads

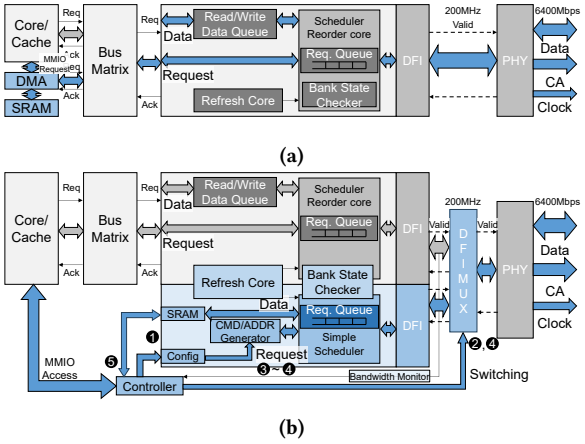


Figure 8: Architectural comparison of PIM control methods. (a) Request-based DMA approach relying on the conventional memory controller. (b) Proposed PDiMC architecture that directly issues DRAM commands using lightweight PIM-specific logic. Blue components indicate added PIM control logic, and blue arrows show the active DRAM control path.

(Figure 8(a)); or (2) by directly generating and sending DRAM commands to the DRAM device, bypassing the MC altogether (Figure 8(b)). The latter provides finer control and better performance for tightly ordered PIM operations. The choice between the two, however, is not only a matter of performance: where the arbitration point is placed also determines whether PIM traffic can be isolated at all.

Request-based control suffers from a key limitation: the MC's internal scheduling, designed for maximizing bandwidth, may reorder RD and WR requests, violating the strict sequencing required by PIM. Prior work [22] addresses this limitation by enforcing memory request order at the DRAM command level, where the next memory request is issued only after the previous one has completed. Although this method reduces latency compared to software-level fencing, it limits opportunities for performance optimizations, such as row pre-activation, and does not fully utilize the potential of the PIM architecture. This limitation is not specific to the method above, but inherent to request-based control in general. Any controller that arbitrates above the MC's scheduling logic must rely on the MC to reach DRAM, and therefore cannot exclude requests that the MC admits from other sources. Ordering and isolation are thus the same problem seen from two sides, and neither can be settled upstream of the reorder logic.

To address these limitations, we propose a PIM-DMA-integrated Memory Controller (PDiMC), illustrated in Figure 8(b), which places arbitration below the reorder logic by pairing a lightweight PIM controller with a DFI multiplexer (MUX). Between the MC and the DRAM device, DFI [44] is the only boundary that is standardized, vendor-neutral, and accessible in the digital domain. Inserting arbitration at the DFI therefore leaves the MC, the DRAM device, and the JEDEC protocol untouched. The lightweight controller thus resides in the same hierarchy as the conventional MC, and the two share the PHY through the MUX.

The lightweight controller generates load/store requests and corresponding DRAM commands for PIM operations and issues the resulting commands directly, independent of host control. Because the controller issues commands in an optimized sequence at the command level, it can enforce precise ordering and pre-open specific rows and banks for improved latency and throughput.

The DFI MUX arbitrates between PIM and non-PIM traffic without hazards, even when the two are interleaved on the memory channel. During PIM execution, the conventional MC's DFI path is deselected, so a competing command has no route to the device, and PDiMC holds full control of the interface.

The DFI MUX is straightforward to implement, with minimal area and timing overhead, and using a MUX for DFI arbitration is standard practice in memory subsystem design (e.g., for DFI mode transitions [23] or PHY testing [10]). The DFI MUX operates entirely within the digital domain at a significantly reduced frequency compared to the memory pin speed, since the PHY absorbs serialization to the DDR pin rate. For example, with a 1:4 ratio between CK and the DFI clock, targeting LPDDR5-6400 requires the DFI MUX to operate at only 200 MHz. Moreover, because DRAM timing is deterministic, the DFI command and data paths operate as a fixed-latency pipeline and do not rely on a FIFO-like valid-ready handshake. The absence of such a handshake inherently simplifies the task of ensuring correctness. The multiplexer's integrity on the write path is easily secured because the enable (dfi_wrdata_en) signal is unidirectional and does not require an acknowledgment. For the read path, the multiplexer's integrity can be secured by ensuring that the number of issued enable signals (dfi_rddata_en) matches the number of received valid signals (dfi_rddata_valid). This matching is feasible because the PHY guarantees a one-to-one correspondence between the valid and enable signals.

Despite bypassing the MC, PDiMC costs far less than reimplementing it, because PIM traffic is deterministic. Once an instruction is fixed, the command sequence follows: for example, GEMV expands into repeated bulk RD and WR operations that load input vectors, trigger the MAC computations, and store output vectors back to DRAM. Unlike conventional MCs, which manage individual DRAM commands and typically require over 32 request queue entries, PDiMC represents complex command sequences (e.g., mode changes or GEMV operations) as single abstract entries, using an internal FSM to issue the corresponding DFI transactions in order. To support parameterized operations such as GEMV execution and result handling, each PDiMC entry includes a start-address and a size field, which the FSM uses to generate the necessary ACT/RD/WR/PRE sequences. This abstraction enables the use of a compact 8-entry request queue without performance degradation, as verified through simulation. The cost is therefore close to zero: the lightweight controller needs neither reordering logic nor a deep queue.

Overall, PDiMC offers a practical, low-overhead, and scalable architecture for integrating PIM into real-world systems. Because the PHYs of other standard DRAM technologies (e.g., HBM and GDDR) utilize the same DFI, as referenced in [13], the PDiMC concept of using a lightweight MC and a DFI MUX can be applied to any JEDEC-standard PIM device.

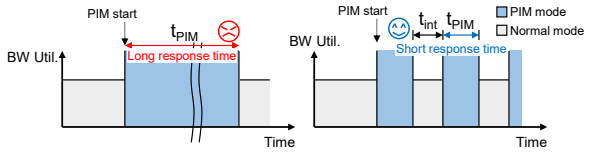


Figure 9: Effect of PDiMC coordination. Interval insertion (right) reduces CPU request latency compared to the baseline.

4.2 Operation Flow and Hardware Architecture of PDiMC

PDiMC is designed to efficiently operate LPPIM by directly interfacing with the DRAM PHY to issue DRAM commands. The PHY interface is shared between the conventional MC and the PDiMC logic, using a DFI MUX to switch command sources. PDiMC receives an input vector along with configuration parameters such as the sizes of the input and output vectors, then generates both memory request sequences and DRAM command sequences to execute the GEMV operation efficiently.

To support this execution model, PDiMC is composed of the following key components: **Request generator** generates a sequence of load/store requests based on the operation type and data size, then forwards them to the lightweight internal controller. **Command generator** converts each memory request into a set of DRAM commands and issues them directly to the PHY interface. **SRAM buffers** serve as temporary storage for input and output vectors used in the GEMV operation. **Refresh core and bank state checker**, which is shared with the conventional MC, monitors refresh timing and the open/close status of each DRAM bank to ensure safe command issuing. **Control and configuration logic** manages the initialization, control flow, and operational state transitions of PDiMC.

The operation flow of PDiMC proceeds as follows, with each step's corresponding path illustrated in Figure 8(b).

1. **Initialization:** The host loads the input vector into the internal SRAM and configures execution parameters, including vector dimensions and the base address of the weight matrix.
2. **PHY Handover:** Triggered by the start configuration bit, the DFI MUX switches control from the conventional MC to PDiMC, initiating isolated PIM execution once pending non-PIM requests are drained.
3. **Command Orchestration:** PDiMC's internal FSM autonomously generates the strict DRAM command sequences required to: (1) dispatch instructions and enter HAB-PIM mode, (2) execute the GEMV operation via bulk RD/WR commands, and (3) safely revert to single-bank (SB) mode upon completion.
4. **Result Handling:** The computed output vectors are fetched from DRAM into the output SRAM. Subsequently, the DFI MUX restores PHY control back to the conventional MC.
5. **Post-Processing:** The host asynchronously retrieves the finalized output vectors from the SRAM to complete the inference step.

This architecture allows PDiMC to handle memory transactions and PIM execution in a tightly coordinated manner, minimizing host involvement while maximizing throughput for memory-bound operations such as GEMV.

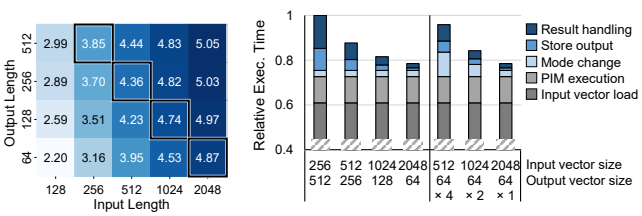


Figure 10: (a) PDiMC GEMV speedup heatmap over baseline. (b) Execution time breakdown for a fixed workload, comparing varying vector dimensions (left four, black-boxed in (a)) versus multiple smaller GEMV blocks (right three).

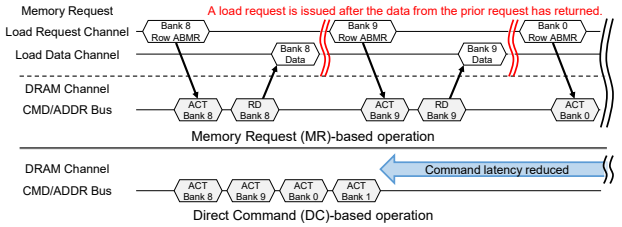


Figure 11: DRAM command-level optimization for mode change. Direct command issuing feature for the mode change sequence on PDiMC reduces mode change latency.

4.3 Concurrent Use of LPPIM for Normal and PIM Requests

In host systems that simultaneously handle both AI and non-AI workloads (e.g., mobile SoCs integrating NPUs and CPUs), it is crucial to ensure that the performance of non-AI tasks is not compromised. When an NPU utilizes LPPIM for accelerating AI operations, prolonged occupation of all memory channels by PIM tasks can significantly degrade the performance of conventional CPU workloads. For instance, consider a GEMV operation where the (input vector size, output vector size) is (2048, 2048), using 16-bit precision weights. If this operation is performed on LPPIM across 4 memory channels, the total processing time is approximately 55 us. During this period, CPU-side tasks requiring DRAM access may be forced to stall, which can severely impact system-level QoS.

To address this issue, PDiMC must be equipped with a scheduling strategy that enables coordinated handling of both PIM operations and normal memory requests. We propose a method in which PDiMC divides a large GEMV operation into smaller GEMV block operations, each consuming t_{PIM}, and inserts short idle intervals (t_{int}) between block executions. These intervals create windows during which normal DRAM transactions from the CPU can be serviced, as illustrated in Figure 9.

To identify an effective GEMV block size, we empirically evaluated performance and system impact for various configurations (see Figure 10(a)). A (2048, 512) GEMV block offers high PIM throughput but causes long delays (longer t_{PIM}) for CPU memory access. On the other hand, reducing the block size shortens these delays (shorter t_{PIM}) but increases the frequency of mode changes, resulting in additional overhead. This trade-off highlights the need for a careful balance between PIM efficiency and system responsiveness.

Breaking down a GEMV operation into several smaller GEMV blocks results in shorter t_{PIM} for each block but can lead to longer overall execution time (see the right graph in Figure 10(b)). This increased execution time stems from the higher number of mode changes. Therefore, there is a sweet spot for the GEMV block size where the system can achieve speedups for both AI and non-AI workloads.

As shown in the left graph in Figure 10(b), even for the same GEMV block size, execution time may vary depending on the relative sizes of the input and output vectors. This asymmetry arises from the different dataflow of how input and output vectors are processed: input vectors are broadcast to all PIM units, while output vectors must be collected individually from the corresponding banks in LPPIM. This effect can be observed in the linear increase in the time consumed for storing the output and handling the result as the output vector size increases. Therefore, configurations with large input vectors and smaller output vectors, such as (2048, 64), are preferable, as they reduce data collection overhead while maintaining performance. Based on this observation, we adopt the (2048, 64) configuration as our default GEMV block size for coordinated scheduling.

4.4 Host-Side Control Method for PDiMC

The programming model for PDiMC follows a structure similar to prior works on near-data processing architectures [8, 24]. In our model, the host, typically an NPU executing LLM inference, offloads GEMV operations to PDiMC while continuing to handle non-LLM tasks asynchronously.

PDiMC exposes an MMIO interface to the host. Both the input/output SRAM regions and the configuration register space are mapped into the host's physical address space. This mapping allows the host to write input vectors, read output vectors, and set execution parameters (e.g., vector size, start address, execution flags) through standard load/store operations. The interface can be implemented using a lightweight AXI slave mapped onto the host IO fabric.

To facilitate dynamic coordination with normal DRAM traffic, the PDiMC device driver monitors system throughput and sets the minimum interleaving interval t_{int} periodically at runtime while the execution time of a single PIM chunk (2048, 64), t_{pim}, remains constant at 7.4 us. At the system level, the performance counters integrated into PDiMC characterize the memory transactions (e.g., memory ACT/RD/WR counters). The device driver can access these counters and dynamically change the ratio between normal and PIM execution by adjusting t_{int}. This adaptive interval helps maintain QoS for non-AI tasks while maximizing the utilization of LPPIM. The datapath operation of PDiMC is independent of changes to t_{int}, ensuring that modifying t_{int} does not affect the correctness of the computation.

When the lightweight MC of PDiMC handles PIM requests, its behavior resembles that of an in-order DMA module operating independently of the CPU. The lightweight MC is non-preemptible within a chunk. Rather than aborting an ongoing PIM operation, it completes the current chunk first. When interleaving is enabled, it then yields the PHY for t_{int} before proceeding to the next chunk, as in Figure 9(b). Any regular DRAM access issued in the meantime is

Table 2: Used System Configuration and Timing parameters

CPU Parameters	Values
Number of Cores	4 CPU Cores, 1 NPU
Cacheline size	32 B
CPU Frequency & NPU Performance	2.4 GHz, 4.8 TOPS
L1/L2/L3 size	32 KB/256 KB/8 MB
L1/L2 type & latency	Private, 1.6/6.4 ns
L3 type & latency	Shared, 20ns
LPPIM Parameters	
Number of Channels & Ranks	4 CHs, 1 Rank
Data Pin frequency	6,400 Mbps
Clock frequency (1/tCK)	800 MHz
tRC, tRCD, tRAS	61.25, 18.75, 42.5 ns
tRPpb, tRPab	18.75, 21.25 ns
tCCD_S, tCCD_L	2 tCK, 4 tCK
tRRD_S, tRRD_L	2 tCK, 4 tCK
PIM execution to PIM execution	4 tCK
I/O GRF size per PIM block	2K-bit/2K-bit
PDiMC & MC Parameters	
MC Scheduling Policy	FR-FCFS, Open-page
Normal MC queue depth	32-entry
Lightweight MC queue depth	8-entry

serviced during this interval. With interleaving disabled, the chunks are issued back-to-back and PDiMC holds the PHY until the GEMV completes.

While the low-level execution model inside LPPIM is beyond the scope of this work, we follow existing PIM programming approaches [33] for instruction loading and data movement. The memory region allocated for weight data stored in LPPIM is configured as a cache-bypass region (such as write-combine on x86) to avoid cache coherence issues with the CPU.

4.5 Additional Optimizations in PDiMC

Command-Level Optimization for Mode Changes: Switching between SB and HAB modes requires issuing a predefined ACT command sequence. When the switch is based on memory requests, strict ordering is only guaranteed if each request completes before the next one is issued, incurring significant latency (see Figure 11). Similarly, as shown in Figure 5, row parking requires 16 load requests to precharge and activate a dummy row across all banks, causing unnecessary data path usage and command overhead.

To address these inefficiencies, PDiMC integrates a lightweight command sequencer that issues DRAM commands directly, bypassing the memory request generator. For mode changes, the sequencer issues ACT commands to four banks sequentially, reducing mode change latency to 4× tRRD. For row parking, the sequencer directly issues an All-Bank Precharge (PREA) command – normally inaccessible via user-level memory requests – ensuring that all banks are closed before entering PIM mode, without generating dummy traffic.

5 Experimental Setup

We evaluated the proposed PDiMC architecture in simulated environment using a modified version of Ramulator [30], comparing it against the baseline (a conventional non-PIM system) and various alternative PIM configurations targeting LPPIM. Our evaluation includes both systems without PIM support and systems using different levels of PIM control from software-managed to fully optimized hardware-managed solutions.

Table 3: Evaluated Configurations

Config.	PIM support	PDiMC	Note
Baseline	No	No	Baseline system
SW-PIM	Yes	No	Using conventional memory fence [33]
Orderlight	Yes	No	Using Orderlight memory fence [41]
PIM-DMA	Yes	No	Dedicated DMA support [22, 27, 28, 36, 45]
MR-PIM	Yes	Yes	Issuing memory request
DC-PIM	Yes	Yes	Issuing DRAM command directly with optimization as in Section 4.5
SW-PIM NF	Yes	No	**SW-PIM** without fence Showing theoretical performance upper bound Functionally incorrect

Table 4: Matrix dimensions of each layer for LLaMA 1B, Pythia, and LLaMA 3B

# of layers	LLaMA 1B		Pythia		LLaMA 3B	
	16		24		28	
	i_dim	o_dim	i_dim	o_dim	i_dim	o_dim
qkv_proj	2048	3072	2048	6144	3072	5120
o_proj	2048	2048	2048	2048	3072	3072
up_ffn	2048	8192	2048	8192	3072	8192
gate_ffn	2048	8192			3072	8192
down_ffn	8192	2048	8192	2048	8192	3072
Default GEMV block	2048	64	2048	64	1024	128

Evaluated Configurations. We define the following seven system configurations as in Table 3.

Workloads. We extracted LLM memory traces from LLaMA-3.2-1B, LLaMA-3.2-3B [40], and Pythia-1.4B [2] running on llama.cpp [11] using Intel Pin [39]. We assumed a single-batch configuration in which QoS is critical, such that the projection and FFN layers (denoted by the _proj and _ffn suffixes, respectively) are computed via GEMV operations. Details of the acceleration target layers for the three LLMs are presented in Table 4. For LLaMA-3.2-3B, we used a GEMV block of (1024, 128) instead of the default (2048, 64), as the input/output dimensions of its GEMV-based layers are not evenly divisible by the default GEMV block. The alternative block shape was chosen to preserve an equivalent GEMV block size as shown in Figure 10(a). The sequence length was set to 512 to reflect realistic mobile LLM usage patterns [61].

We excluded attention layers from PIM acceleration for two reasons. In LLaMA-based models, GQA means that attention computations do not fall into GEMV operations, making PIM acceleration

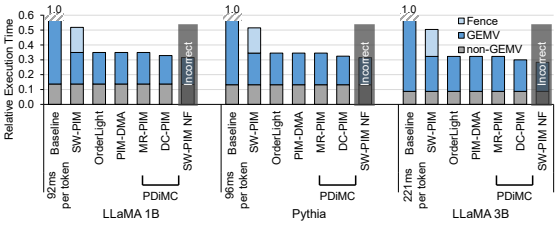


Figure 12: Relative execution time breakdown for LLM inference across various PIM configurations.

ineffective. In Pythia, although MHA is employed, the performance benefit is marginal at short sequence lengths.

For **SW-PIM** and **Orderlight**, we generated traces by simulating the PIM control in software and manually inserting fences into the trace post-processing stage. For the **SW-PIM NF** (NoFence) configuration, we removed the fence instructions, which may lead to functionally incorrect behavior but allow performance bounding.

To evaluate performance under multiprogrammed conditions, we constructed workloads using benchmarks from the SPEC CPU 2006 [14]. Each experiment utilized a four-application mix, with sampled 100M instructions. We designed two distinct mixes, each balanced according to memory intensity characteristics (one high, two medium, and one low), based on the method described in [35]:
spec-mix1: lbm, cactusADM, sphinx3, and xalancbmk.
spec-mix2: mcf, soplex, astar, and bzip2.

Experimental Modes. We conducted two main experiments:

(1) **LLM-only execution:** We ran the LLM inference in isolation across all seven configurations (**Baseline**, **SW-PIM**, **Orderlight**, **PIM-DMA**, **MR-PIM**, **DC-PIM**, **SW-PIM NF**) to evaluate the performance impact of PIM.
(2) **LLM + Multi-programmed workloads:** We co-executed LLM inference with SPEC workloads under three configurations (**Baseline**, **MR-PIM**, and **DC-PIM**). We measured the effect of PDiMC and evaluated coordination strategies using configurable interleaving intervals (t_{int}) and GEMV block sizes. **SW-PIM**, **Orderlight**, and **PIM-DMA** were excluded due to their reliance on conventional memory controllers, which lack the capability to isolate concurrent memory traffic during PIM execution as explained in Section 3.2.

Hardware Overhead Estimation. PDiMC was implemented on a Xilinx U280 [60] platform using S-PIM, with its RTL designed based on the DFI interface. S-PIM is only solution to evaluate JEDEC-compliant protocol based PIM, so we choose PHY-only option with U280 IP generator to implement simple MC and DFI mux to connect PDiMC. Functional verification and experiment were conducted using Cadence Xcelium, Synopsys Synplify [31] and Xilinx Vivado [9]. The RTL was then modified to target ASAP 7nm [4] technology, using SRAM modules provided in the corresponding design kit. Additionally, the design was adapted to comply with the DFI 5.1 specification [44] to support LPDDR5 operation.

The finalized RTL was synthesized and placed-and-routed using Synopsys Design Compiler NXT [57] and ICC2 [56] to generate the GDS. The resulting PDiMC block supports 200MHz operation on SS (slowest) corner for the DFI interface, enabling a 4:1 ratio to achieve 6400Mbps. The total logic area was 0.060mm^2, and the SRAM area, which comprises two 16KB SRAMs for handling (2048, 64) GEMV blocks, was 0.008mm^2. This module consumes average 48mW on 200Mhz for given LLM workloads by PrimePower [58].

6 Evaluation

Figure 12 presents the speedup achieved by adopting a PIM architecture. As various optimizations are applied, the system progressively reduces overhead and improves performance. The software-controlled PIM system (**SW-PIM**) achieves 1.93×, 1.94×, and 1.98× speedup over the **Baseline**, mainly due to 4.07×, 4.08×, and 3.87× acceleration in GEMV operations, for LLaMA 1B, Pythia, and LLaMA 3B, respectively. However, a substantial portion of the remaining runtime – about 44% of **SW-PIM**'s GEMV execution time on average – is consumed by memory fences, which are necessary to maintain command ordering during mode changes. Replacing conventional memory ordering with more efficient approaches yields additional gains. Specifically, employing a lightweight ordering primitive such as **Orderlight** or using a PIM-aware DMA engine (**PIM-DMA**) leads to a further 48% (LLaMA 1B), 49% (Pythia), and 56% (LLaMA 3B) speedup over **SW-PIM**. This speedup stems from effectively mitigating fence overhead. Notably, the performance achieved by both **Orderlight** and **PIM-DMA** is comparable to that of the hardware-based memory-request-level PDiMC (**MR-PIM**), indicating that minimizing fence latency – whether through software or hardware – is the key to unlocking PIM efficiency.

By implementing PDiMC, the system further achieves speedups over existing architectures. Moving from **MR-PIM** to **DC-PIM** provides a further 6.9% speedup on average by enabling finer-grained command scheduling and reducing redundant ACT/PRE commands. This brings the cumulative speedup over **SW-PIM** to 58% (LLaMA 1B), 59% (Pythia), and 68% (LLaMA 3B). These combined optimizations bring the system to within 4.4% on average of **SW-PIM NF**, a configuration that skips fences entirely but sacrifices correctness.

While **Orderlight** and **PIM-DMA** achieve performance comparable to **MR-PIM** for isolated LLM inference, their reliance on conventional memory controllers presents a key limitation as explained in Section 3.2. Specifically, **Orderlight** and **PIM-DMA** lack a coordination mechanism for safely interleaving PIM and non-PIM requests in multi-programmed environments. As a result, they are not suitable for systems requiring concurrent access to LPPIM from multiple hosts, as shown in Figure 13(a).

Offloading GEMV operations to PDiMC not only accelerates LLM inference but also improves non-LLM task performance. Figure 13(a) evaluates system performance under concurrent LLM inference and SPEC benchmark workloads, where PDiMC continues to provide significant gains. With **DC-PIM**, LLM throughput improves by 2.8× (**spec-mix1**) and 2.2× (**spec-mix2**), while non-LLM IPC also improves by 13.5% and 0.7%, respectively, indicating that reduced memory contention benefits all workloads sharing the DRAM channel. These results are averaged across all three evaluated models (LLaMA 1B, Pythia, and LLaMA 3B).

The balance between LLM acceleration and system throughput can be tuned via t_{int}, as shown in Figure 13(b). Smaller values aggressively utilize PIM mode, maximizing LLM throughput at the cost of non-LLM IPC, while larger values favor normal mode access.

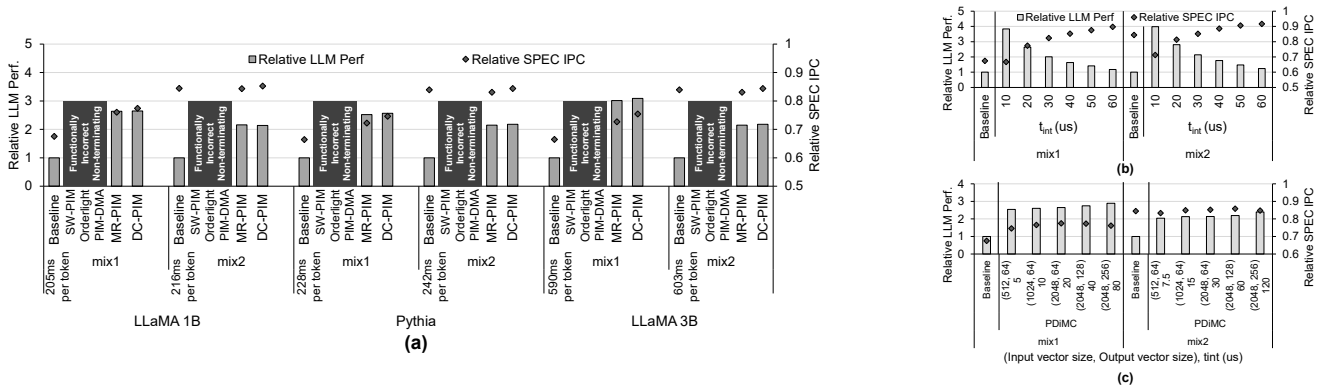


Figure 13: Impact of PIM control methods and interleaving on co-running workloads. (a) Relative performance of LLM and host workloads when executed concurrently under various PIM configurations. (b) Effect of t_{int} and (c) GEMV block on the performance trade-off between LLM and CPU workloads. Results in (b) and (c) are based on LLaMA 1B experiments.

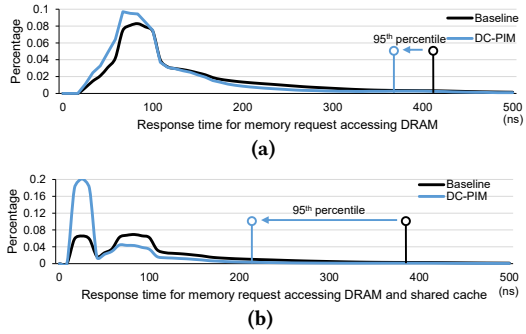


Figure 14: Response time distribution for non-LLM memory requests accessing (a) only DRAM and (b) both DRAM and shared cache under spec-mix1 and LLaMA 1B.

A moderate setting (e.g., t_{int} = 20–30 µs) achieves over 2× LLM speedup while maintaining more than 80% of baseline IPC.

Similarly, Figure 13(c) shows that GEMV block size and t_{int} jointly affect this trade-off. Our results show (2048, 64) is optimal for **spec-mix1** and (2048, 128) for **spec-mix2**, indicating that the optimal configuration depends on the running environment.

By isolating bandwidth-heavy LLM memory traffic, PDiMC mitigates queuing delays and cache contention, improving responsiveness for latency-sensitive tasks. Figure 14 shows the latency distribution of non-LLM memory requests, measured at the 95th percentile as a proxy for tail latency. With **DC-PIM**, tail latency decreases by 12.2% for DRAM-bound requests and by 42.1% when including shared L3 cache hits, demonstrating that offloading GEMV to PIM benefits the memory subsystem as a whole.

7 Related Work

There have been various efforts to integrate PIM systems with host processors or accelerators. UPMEM [7], a real-world PIM platform, uses system software and a runtime library [3] to configure and execute PIM kernels. However, sequential execution between host and PIM kernels limits performance [16, 32], making direct PIM command issue from the host a more practical alternative [33, 34].

Orderlight [41] proposed a memory ordering primitive for PIM, but assumes fine-grained DRAM control incompatible with the coarse-grained mode changes required by JEDEC-standard PIM. Similarly, [49] addresses PIM memory consistency, but is limited to bulk-bitwise architectures with separate PIM commands, restricting applicability to JEDEC-compliant systems. [62, 63] proposed MC-level scheduling to balance conventional memory accesses and PIM operations. However, their reliance on new commands and fine-grained PIM control limits practicality for real-world systems.

For GEMV acceleration, [26, 54] explored DRAM-based PIM but required modifications to both the DRAM protocol and memory controller. Building on these, [33, 34] introduced practical PIM architectures that efficiently perform GEMV.

8 Conclusion

We have addressed the challenge of maintaining strict memory command ordering in PIM systems by proposing PDiMC, a lightweight, JEDEC-compatible controller capable of issuing both PIM-aware memory requests and DRAM commands. PDiMC not only enables high-throughput LLM execution but also reduces average and tail latency for non-PIM workloads, offering a scalable and QoS-aware path toward practical PIM integration in general-purpose SoCs.

References

[1] Anthropic. [n. d.]. The Claude 3 Model Family: Opus, Sonnet, Haiku.
[2] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*. PMLR, 2397–2430.
[3] Jinfan Chen, Juan Gómez-Luna, Izzat El Hajj, Yuxin Guo, and Onur Mutlu. 2023. SimplePIM: A Software Framework for Productive and Efficient Processing-in-Memory. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 99–111. doi:10.1109/PACT58117.2023.00017
[4] Lawrence T Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm FinFET Predictive Process Design Kit. *Microelectronics Journal* 53 (2016), 105–115. doi:10.1016/j.mejo.2016.04.006
[5] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG] https://arxiv.org/abs/2205.14135
[6] DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948
[7] Fabrice Devaux. 2019. The true Processing In Memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. 1–24. doi:10.1109/HOTCHIPS.2019.8875680
[8] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM Acceleration Architecture Leveraging Commodity

DRAM Devices and Standard Memory Modules. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 283–295. doi:10.1109/HPCA.2015.7056040
[9] Tom Feist. 2012. Vivado design suite. *White Paper* 5, 30 (2012), 24.
[10] Johannes Feldmann, Jan Lappas, Mohammadreza Esmaeilpour, Hussien Abdo, Christian Weis, and Norbert Wehn. 2025. Enhanced LPDDR4X PHY in 12 nm FinFET. *arXiv preprint arXiv:2503.11654* (2025).
[11] Georgi Gerganov. [n. d.]. llama.cpp. https://github.com/ggerganov/llama.cpp
[12] Christina Giannoula, Peiming Yang, Ivan Fernandez, Jiacheng Yang, Sankeerth Durvasula, Yu Xin Li, Mohammad Sadrosadati, Juan Gomez Luna, Onur Mutlu, and Gennady Pekhimenko. 2024. PyGim: An Efficient Graph Neural Network Library for Real Processing-In-Memory Architectures. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 8, 3 (2024), 1–36.
[13] Kos Gitchev. 2024. LPDDR, GDDR, and HBM for Auto AI Applications. https://www.jedec.org/sites/default/files/Final_Kos_Gitchev_LPDDR_GDDR_HBM_Auto_AI_Applications_v3.pdf
[14] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. doi:10.1145/1186736.1186737
[15] Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park. 2024. NeuPIMs: NPU-PIM Heterogeneous Acceleration for Batched LLM Inferencing. In *ASPLOS, Volume 3*. 722–737. doi:10.1145/3620666.3651380
[16] Bongjoon Hyun, Taehun Kim, Dongjae Lee, and Minsoo Rhu. 2024. Pathfinding future pim architectures by demystifying a commercial pim technology. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 263–279.
[17] Bruce Jacob, Spencer Ng, and David Wang. 2007. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[18] JEDEC. 2015. High Bandwidth Memory DRAM(HBM1,HBM2).
[19] JEDEC. 2017. DDR4 SDRAM Standard.
[20] JEDEC. 2021. Graphics Double Data Rate (GDDR6) SGRAM Standard.
[21] JEDEC. 2023. LOW POWER DOUBLE DATA RATE (LPDDR) 5/5X.
[22] Shinhaeng Kang, Sukhan Lee, Byeongho Kim, Hweesoo Kim, Kyomin Sohn, Nam Sung Kim, and Eojin Lee. 2022. An fpga-based rnn-t inference accelerator with pim-hbm. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 146–152.
[23] Kishore Kasamsetty and Jeffrey Chung. 2014. Design considerations for LPDDR4/3 PHY and controller sub system. https://www.jedec.org/sites/default/files/Kishor_Kasamsetty_Jeffery_Chung_MF_Aug_2014_copyright.pdf
[24] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, Yeongon Cho, Jin Hyun Kim, Yongsuk Kwon, Kyungsoo Kim, Jin Jung, Ilkwon Yun, Sung Joo Park, Hyunsun Park, Joonho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin Sean Lee. 2021. Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM. *IEEE Micro* 01 (July 2021), 1–1. doi:10.1109/MM.2021.3097700
[25] Byeongho Kim, Sanghoon Cha, Sangsoo Park, Jieun Lee, Sukhan Lee, Shinhaeng Kang, Jinin So, Kyungsoo Kim, Jin Jung, Jong-Geon Lee, et al. 2024. The Breakthrough Memory Solutions for Improved Performance on LLM Inference. *IEEE Micro* (2024).
[26] Byeongho Kim, Jongwook Chung, Eojin Lee, Wonkyung Jung, Sunjung Lee, Jaewan Choi, Jaehyun Park, Minbok Wi, Sukhan Lee, and Jung Ho Ahn. 2020. MViD: Sparse Matrix-Vector Multiplication in Mobile DRAM for Accelerating Recurrent Neural Networks. *IEEE Trans. Comput.* 69, 7 (2020), 955–967. doi:10.1109/TC.2020.2984496
[27] Chang Hyun Kim, Won Jun Lee, Yoonah Paik, Seok Young Kim, and Seon Wook Kim. 2023. BL-PIM: Varying the Burst Length to Realize the All-Bank Performance and Minimize the Multi-Workload Interference for in-DRAM PIM. *IEEE Access* 11 (2023), 81143–81156.
[28] Chang Hyun Kim, Won Jun Lee, Yoonah Paik, Kiyong Kwon, Seok Young Kim, Il Park, and Seon Wook Kim. 2021. Silent-PIM: Realizing the processing-in-memory computing with standard memory requests. *IEEE Transactions on Parallel and Distributed Systems* 33, 2 (2021), 251–262.
[29] Guhyun Kim, Jinkwon Kim, Nahsung Kim, Woojae Shin, Jongsoon Won, Hyunha Joo, Haerang Choi, Byeongju An, Gyeongcheol Shin, Dayeon Yun, Jeongbin Kim, Changhyun Kim, Ilkon Kim, Jaehan Park, Yosub Song, Byeongsu Yang, Hyeongdeok Lee, Seungyeong Park, Wonjun Lee, Seonghun Kim, Yonghoon Park, Yousub Jung, Gi-Ho Park, and Euicheol Lim. 2024. SK Hynix AI-Specific Computing Memory Solution: From AiM Device to Heterogeneous AiMX-xPU System for Comprehensive LLM Inference. In *2024 IEEE Hot Chips 36 Symposium (HCS)*. 1–26. doi:10.1109/HCS61935.2024.10664793
[30] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (Jan. 2016), 45–49. doi:10.1109/LCA.2015.2414456
[31] Pran Kurup and Taher Abbasi. 2012. *Logic synthesis using Synopsys*. Springer.
[32] Dongjae Lee, Bongjoon Hyun, Taehun Kim, and Minsoo Rhu. 2024. PIM-MMU: A Memory Management Unit for Accelerating Data Transfers in Commercial PIM Systems. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 627–642.
[33] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM based on Commercial DRAM Technology: Industrial Product. In *ISCA*. doi:10.1109/ISCA52012.2021.00013
[34] Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Kornijcuk Vladimir, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Jaewook Lee, Donguc Ko, Younggun Jun, Keewon Cho, Ilwoong Kim, Choungki Song, Chunseok Jeong, Daehan Kwon, Jieun Jang, Il Park, Junhyun Chun, and Joohwan Cho. 2022. A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications. In *2022 IEEE International Solid-State Circuits Conference*, Vol. 65. 1–3. doi:10.1109/ISSCC42614.2022.9731711
[35] Sukhan Lee, Kiwon Lee, Minchul Sung, Mohammad Alian, Chankyung Kim, Wooyeong Cho, Reum Oh, Seongil O, Jung Ho Ahn, and Nam Sung Kim. 2018. 3D-Xpath: High-density Managed DRAM Architecture with Cost-effective Alternative Paths for Memory Transactions. In *PACT*. 12 pages. doi:10.1145/3243176.3243191
[36] Won Jun Lee, Chang Hyun Kim, Yoonah Paik, and Seon Wook Kim. 2023. PISA-DMA: Processing-in-Memory Instruction Set Architecture Using DMA. *IEEE Access* 11 (2023), 8622–8632. doi:10.1109/ACCESS.2023.3238812
[37] Cong Li, Zhe Zhou, Size Zheng, Jiaxi Zhang, Yun Liang, and Guangyu Sun. 2024. SpecPIM: Accelerating speculative inference on PIM-enabled system via architecture-dataflow co-exploration. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 950–965.
[38] David Luebke. 2008. CUDA: Scalable Parallel Programming for High-Performance Scientific Computing. In *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro (ISBI)*. 836–838. doi:10.1109/ISBI.2008.4541126
[39] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.
[40] Meta. 2024. Llama3. https://llama.meta.com/llama3/
[41] Anirban Nag and Rajeev Balasubramonian. 2021. OrderLight: Lightweight memory-ordering primitive for efficient fine-grained PIM computations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 298–310.
[42] on-dev-ai bench. 2025. on-device-generative-ai-benchmarks. https://nexa.ai/blogs/on-device-generative-ai-benchmarks/
[43] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[44] DDR PHY ORG. [n. d.]. DFI. https://ddr-phy.org/
[45] Yoonah Paik, Chang Hyun Kim, Won Jun Lee, and Seon Wook Kim. 2022. Achieving the Performance of All-Bank In-DRAM PIM With Standard Memory Interface: Memory-Computation Decoupling. *IEEE Access* 10 (2022), 93256–93272. doi:10.1109/ACCESS.2022.3203051
[46] Jaehyun Park, Jaewan Choi, Kwanhee Kyung, Michael Jaemin Kim, Yongsuk Kwon, Nam Sung Kim, and Jung Ho Ahn. 2024. AttAcc! Unleashing the Power of PIM for Batched Transformer-based Generative Model Inference. In *ASPLOS, Volume 2*. 103–119. doi:10.1145/3620665.3640422
[47] Jaehyun Park, Byeongho Kim, Sungmin Yun, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. 2021. TRiM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory. In *MICRO*.
[48] Sang-Soo Park, KyungSoo Kim, Jinin So, Jin Jung, Jonggeon Lee, Kyoungwan Woo, Nayeon Kim, Younghyun Lee, Hyungyo Kim, Yongsuk Kwon, Jinhyun Kim, Jieun Lee, YeonGon Cho, Yongmin Tai, Jeonghyeon Cho, Hoyoung Song, Jung Ho Ahn, and Nam Sung Kim. 2024. An LPDDR-based CXL-PNM Platform for TCO-efficient Inference of Transformer-based Large Language Models. 970-982 pages. doi:10.1109/HPCA57654.2024.00078
[49] Ben Perach, Ronny Ronen, and Shahar Kvatinsky. 2023. On consistency for bulk-bitwise processing-in-memory. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 705–717.
[50] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. 2000. Memory access scheduling. *ACM SIGARCH Computer Architecture News* 28, 2 (2000), 128–138.
[51] SAITPublic. [n. d.]. PIMSimulator. https://github.com/SAITPublic/PIMSimulator
[52] Minseok Seo, Xuan Truong Nguyen, Seok Joong Hwang, Yongkee Kwon, Guhyun Kim, Chanwook Park, Ilkon Kim, Jaehan Park, Jeongbin Kim, Woojae Shin, Jongsoon Won, Haerang Choi, Kyuyoung Kim, Daehan Kwon, Chunseok Jeong, Sangheon Lee, Yongseok Choi, Wooseok Byun, Seungcheol Baek, Hyuk-Jae Lee, and John Kim. 2024. IANUS: Integrated Accelerator based on NPU-PIM Unified Memory System. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 545–560. doi:10.1145/3620666.3651324

[53] Changmin Shin, Jaeyong Song, Hongsun Jang, Dogeun Kim, Jun Sung, Taehee Kwon, Jae Hyung Ju, Frank Liu, Yeonkyu Choi, and Jinho Lee. 2025. Piccolo: Large-Scale Graph Processing with Fine-Grained In-Memory Scatter-Gather. arXiv:2503.05116 [cs.AR] https://arxiv.org/abs/2503.05116

[54] Hyunsung Shin, Dongyoung Kim, Eunhyeok Park, Sungho Park, Yongsik Park, and Sungjoo Yoo. 2018. McDRAM: Low Latency and Energy-Efficient Matrix Computations in DRAM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2613–2622. doi:10.1109/TCAD.2018.2857044

[55] Jaihyuk Song. 2025. AI Revolution Driven by Memory Technology Innovation. In *2025 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 68. 26–36. doi:10.1109/ISSCC49661.2025.10904790

[56] Synopsys. 2019. IC Compiler II. https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/ic-compiler-ii-ds.pdf

[57] Synopsys. 2022. Synopsys Design Compiler NXT. https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/design-compiler-nxt-ds.pdf

[58] Synopsys. 2022. Synopsys PrimePower. https://www.synopsys.com/implementation-and-signoff/signoff/primepower.html

[59] Gemini Team. 2024. Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.11805 [cs.CL] https://arxiv.org/abs/2312.11805

[60] Xilinx. 2021. Xilinx Alveo U280 Data Center Accelerator Card. https://xilinx.com/products/boards-and-kits/alveo/u280.html

[61] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. 2025. Fast on-device llm inference with npus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 445–462.

[62] Seunghyuk Yu, Hyeonu Kim, Kyoungho Jeun, Sunyoung Hwang, Seongmin Cho, and Eojin Lee. 2025. ComPASS: A Compatible PIM Protocol Architecture and Scheduling Solution for Processor-PIM Collaboration. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. 49–62.

[63] Seunghyuk Yu, Hyeonu Kim, Kyoungho Jeun, Sunyoung Hwang, and Eojin Lee. 2024. Architecting Compatible PIM Protocol for CPU-PIM Collaboration. *IEEE Computer Architecture Letters* (2024).

[64] Sungmin Yun, Kwanhee Kyung, Juhwan Cho, Jaewan Choi, Jongmin Kim, Byeongho Kim, Sukhan Lee, Kyomin Sohn, and Jung Ho Ahn. 2024. Duplex: A Device for Large Language Models with Mixture of Experts, Grouped Query Attention, and Continuous Batching. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1429–1443.

[65] Sungmin Yun, Hwayong Nam, Kwanhee Kyung, Jaehyun Park, Byeongho Kim, Yongsuk Kwon, Eojin Lee, and Jung Ho Ahn. 2024. CLAY: CXL-based Scalable NDP Architecture Accelerating Embedding Layers. In *Proceedings of the 38th ACM International Conference on Supercomputing*. 338–351. doi:10.1145/3650200.3656595

[66] Sungmin Yun, Hwayong Nam, Jaehyun Park, Byeongho Kim, Jung Ho Ahn, and Eojin Lee. 2023. GraNDe: Efficient Near-Data Processing Architecture for Graph Neural Networks. *IEEE Trans. Comput.* (2023), 1–14. doi:10.1109/TC.2023.3283677