

Characterizing Digital DRAM PIM through Modeling and Benchmarking

FARZANA AHMED SIDDIQUE*, DEYUAN GUO*, HUGO ABBOT, KYLE DURRER, MOHAMMADHOSEIN GHOLAMREZAEI, MORTEZA BARADARAN, ETHAN ERMOVICK, ALIF AHMED, ZHENXING FAN, BEENISH GUL, ASHISH VENKAT, and KEVIN SKADRON, University of Virginia, USA

The disparity between processor speed and memory bandwidth has become a growing performance bottleneck, particularly for memory-intensive workloads. Processing-in-Memory (PIM) mitigates this bottleneck by integrating computation directly within DRAM. However, the effectiveness of PIM varies significantly across workloads, architectures, and DRAM technologies, yet it is often assessed using tightly coupled simulators and benchmarks that lack portability and generality. This paper extends *PIMbench* and *PIMeval*—a generalizable benchmark suite and an extensible PIM simulator to support a broader range of workloads, PIM architectures, and DRAM technologies. This evaluation incorporates roofline analysis and a breakdown of intra-memory execution stages to identify PIM-specific bottlenecks and performance scaling limits. The evaluation spans three classes of digital PIM architectures: subarray-level bit-serial, subarray-level bit-parallel, and bank-level bit-parallel. It further demonstrates how internal DRAM parameters such as subarray count and GDL width impact PIM performance. The code is publicly available at: <https://github.com/UVA-LavaLab/PIMeval-PIMbench>

CCS Concepts: • **Hardware** → **Emerging architectures**; • **Computing methodologies** → **Modeling and simulation**; • **General and reference** → **Evaluation**.

Additional Key Words and Phrases: Processing-in-Memory, benchmark, simulator, roofline

ACM Reference Format:

Farzana Ahmed Siddique, Deyuan Guo, Hugo Abbot, Kyle Durrer, Mohammadhosein Gholamrezaei, Morteza Baradaran, Ethan Ermovick, Alif Ahmed, Zhenxing Fan, Beenish Gul, Ashish Venkat, and Kevin Skadron. 2026. Characterizing Digital DRAM PIM through Modeling and Benchmarking. 1, 1 (May 2026), 25 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

For decades, Moore’s Law [53] and Dennard Scaling [11] sustained computing performance by increasing transistor density without increasing power consumption. As these trends have slowed [18, 31], traditional performance scaling

*Equal contribution.

Extension of Conference Paper. This manuscript extends our prior work published in the *IISWC 2024* [63]. This submission contains 30% new material: we extend PIMbench with BFS, PR, Prefix Sum, Random Forest, and ResNet (including distinct Prefix Sum implementations for bit-serial vs. bit-parallel PIM); extend PIMeval to model multiple DRAM technologies, PIM-PIM communication, and per-kernel invocation overhead; add roofline analysis spanning both bandwidth and compute throughput (achieved vs. peak); provide cross-DRAM-technology analysis; incorporate Aquabolt validation; introduce load balancing for bit-parallel PIM; and add a GEMM-based optimization study highlighting opportunities for bank-level PIM.

Authors’ Contact Information: Farzana Ahmed Siddique, farzana@virginia.edu; Deyuan Guo, dg7vp@virginia.edu; Hugo Abbot, drt3sm@virginia.edu; Kyle Durrer, kmd2zjw@virginia.edu; Mohammadhosein Gholamrezaei, uab9qt@virginia.edu; Morteza Baradaran, rgq5aw@virginia.edu; Ethan Ermovick, keg9ve@virginia.edu; Alif Ahmed, alifahmed@virginia.edu; Zhenxing Fan, fjy3ws@virginia.edu; Beenish Gul, bg9qq@virginia.edu; Ashish Venkat, venkat@virginia.edu; Kevin Skadron, skadron@virginia.edu, University of Virginia, Charlottesville, Virginia, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

has become unsustainable. On top of that, the growing disparity between processor speed and memory bandwidth—commonly referred to as the memory wall [78], is further exacerbating the problem. As shown in Figure 1a, for AMD server class CPUs, performance increases by $\sim 4\times$ from Naples to Genoa, while memory bandwidth improves $\sim 2\times$. For NVIDIA GPUs (Figure 1b), performance improves by $\sim 93\times$ from Pascal to Hopper, whereas, memory bandwidth increases by $\sim 5\times$. This is especially problematic for memory-intensive workloads, such as those in machine learning, databases, and linear algebra, where frequent memory accesses incur high latency and energy costs [55, 62]. These workloads often exhibit low compute intensity and poor locality; data is frequently fetched only to be discarded after minimal use, compounding inefficiency [22].



(a) Performance and bandwidth scaling for AMD EPYC CPUs. While performance increases by $\sim 4\times$ from Naples to Genoa, memory bandwidth improves by $\sim 2\times$.

(b) Performance and bandwidth trends for NVIDIA GPUs. Performance improves by $\sim 93\times$ from Pascal to Hopper, while bandwidth grows by $\sim 5\times$.

Fig. 1. Performance increases at a much faster rate than memory bandwidth. Data points are extracted from vendor datasheets, selecting the highest-end model from each generation [69, 74]. Area of the circles and squares indicate the scaling of performance and bandwidth relative to the baseline - Naples for CPU and Pascal for GPU.

To address this, researchers have proposed moving computation closer to data, even in the memory itself. First introduced by Stone in 1970 [67], Processing-in-Memory (PIM) embeds simple compute units within DRAM to reduce data movement via in-place computation. Both academic [17, 27, 30, 34, 42, 43, 46, 60, 62, 76] and industrial efforts [19, 37, 38] have demonstrated PIM's potential. However, the benefits of PIM vary significantly across workloads, architectural choices, and underlying memory technologies [22], making it essential to systematically characterize where and how PIM is most effective. A key challenge in this space has been the lack of generalizable evaluation infrastructure. Most PIM proposals, whether academic or industrial, rely on architecture-specific benchmarks and simulators, often designed to highlight the strengths of the proposed system while overlooking potential limitations. While prior efforts have explored simulation [4, 79, 80] and benchmarking [23, 25, 42] for PIM, cross-architecture comparison remained challenging due to the lack of a unified and portable framework. *PIMbench* and *PIMeval* [63] aimed to bridge this gap by providing an extensible benchmark suite and modeling infrastructure.

This paper adds roofline analysis [75], a principled model that disentangles the effects of memory bandwidth and compute throughput, and highlights the gap between achieved and peak performance for a workload at a given operational intensity on a specific architecture. We further decompose PIM kernel execution time to isolate intra-memory data movement overheads (Figure 9), revealing architecture-specific bottlenecks, such as ALU underutilization—that help explain when PIM vs. conventional processors, such as GPUs, are preferable. For example, roofline analysis shows why PIM excels at AXPY, provides slightly better performance for GEMV, and under-performs GPUs on compute-intensive workloads such as matrix-matrix multiplication. Additionally, we extend *PIMbench* to include a broader set of workloads, including deep learning models with residual connections (e.g., ResNet), control-intensive models (e.g., random forest), data-parallel primitives (e.g., prefix sum), and graph analytics (e.g. BFS and PR). We also expand *PIMeval* to support emerging DRAM technologies such as HBM, LPDDR, and GDDR. Our analysis (Section 10) reveals that, while subarray-level PIM performance remains largely unaffected by DRAM JEDEC standard when considering

computation alone, all PIM variants benefit from HBM’s significantly higher memory bandwidth in end-to-end execution. Furthermore, bank-level PIM sees gains in throughput with a wider global data line (GDL) interface that allows more data to be fetched per column access.

This paper evaluates PIM as a co-processor alongside CPUs and GPUs. To this end, we extend the PIMbench and PIMeval framework [63] by adding modeling support for diverse DRAM technologies, and extending the set of benchmarks (Section 7). This paper considers three potential locations for placing digital PIM processing: subarray-level bit-serial [20, 46], subarray-level bit-parallel [42], and bank-level bit-parallel [12, 30]. We also add roofline analysis to provide a principled performance–bandwidth perspective (Section 8) for these PIM architectures, identifying inflection points based on operational intensity where CPUs or GPUs begin to outperform the PIM architectures studied in this work. Furthermore, we analyze how DRAM variants such as DDR and HBM, as well as changes to internal DRAM architecture—such as GDL width—affect PIM performance (Section 10). Collectively, these contributions lay the foundation for systematic, cross-platform evaluation of PIM.

2 Related Work

PIM Modeling and Simulation. DRAMsim3 [47] and Ramulator [36] are cycle-accurate DRAM simulators that do not support PIM execution, although both can model the timing of PIM-specific DRAM reads/writes [12, 30]. PIMSim [80] annotates programs with PIM instructions that invoke a performance model built around CPU/GPU-like cores, which is suitable only for bank-interface or near-memory PIM and not subarray-level PIM. MultiPIM [83] aims to be a flexible PIM framework but assumes a shared ISA for host and PIM cores, restricting exploration of heterogeneous PIM instruction sets. gem5 [6] is a CPU/GPU simulator with detailed conventional memory modeling, and has been used in PIM [59], but these extensions are specialized and not easily reusable for diverse PIM designs. Across these tools, adding a new benchmark typically requires low-level programming. In contrast, the proposed PIM API enables applications to be written in a high-level language, abstracting low-level hardware details and providing a portability layer across different PIM architectures.

Benchmark Suites for PIM. PIMbench is not the first benchmark suite for PIM, but it is the first suite that is not

Table 1. PIMbench Suite.

Domain	Benchmark	Memory Access Pattern			Execution Type	Dataset
		Sequential	Strided	Random		
Linear Algebra	Vector Addition	✓			PIM	2,147,483,648 32-bit INT
	AXPY	✓			PIM	263,435,456 32-bit INT
	Matrix-Vector Mul. (GEMV)	✓			PIM	1K 4, 096 × 4, 096 32-bit INT
	Matrix-Matrix Mul. (GEMM)	✓			PIM	16, 384 × 4, 096 and 4, 096 × 16384 32-bit INT
Parallel Scan	Prefix Sum	✓			PIM + Host	1,073,741,824 32-bit INT
Sort	Radix Sort	✓		✓	PIM + Host	134,217,728 32-bit INT
Cryptography	AES	✓			PIM	1,073,741,824 Bytes
Graph	Triangle Count	✓			PIM	155,331 nodes and 11,128,172 edges [81]
	Breadth First Search (BFS)	✓		✓	PIM + Host	4,039 nodes and 88,234 edges [45]
	Page Rank (PR)	✓		✓	PIM + Host	4,039 nodes and 88,234 edges [45]
Database	Filter-By-Key	✓		✓	PIM + Host	2,147,483,648 key-value pairs
Image Processing	Histogram	✓			PIM	1.4 × 10 ⁹ 24-bit .bmp
	Brightness	✓			PIM	
	Image Downsampling	✓			PIM	
Supervised Learning	k-Nearest Neighbors (k-NN)	✓		✓	PIM + Host	6,710,886 2D points
	Linear Regression (LR)	✓			PIM	1,073,741,824 2D points
	Random Forest	✓			PIM + Host	Synthetic dataset with 2k Decision Trees
Unsupervised Learning	k-means	✓			PIM	134,217,728 2D points, $k = 20$
Neural Network	VGG	✓	✓		PIM + Host	64, 224X224X3 images
	ResNet	✓	✓		PIM + Host	

tied to a specific PIM architecture. Prior work has introduced several benchmark suites targeting UPMEM [23, 25, 50] that use the UPMEM SDK [20]. This tight coupling of benchmark suites to proprietary PIM products may later become inaccessible. One might expect that UPMEM-based simulators [33] could serve as a fallback, but these simulators also

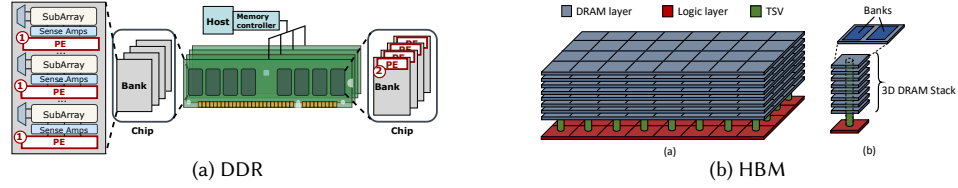


Fig. 2. DRAM Variants

depend on the SDK and are therefore similarly affected. In contrast, InSituBench [42] for Fulcrum does not suffer from an SDK availability issue, but adapting it to other PIM architectures requires rewriting the entire benchmark suite in low-level micro-op. IRAM proposed a suite of five benchmarks [21], but it is no longer available; a version of this suite's histogram benchmark was also included in the Phoenix benchmark suite for map-reduce processing [82]. BLIMP [12] introduces compiler analysis to map the Phoenix and SPECcpu benchmarks to a bank-level PIM architecture, i.e., a simple RISC-V core per bank, but this approach is tied to a fully-featured ISA such as RISC-V. Extending that compiler analysis to other PIM architectures is a promising direction for future work.

As summarized in Table 1, PIMbench spans from simple building blocks—such as vector addition and AXPY—to complex workloads that chain multiple kernels, require data re-layout between stages, and exercise PIM–host communication, such as VGG and ResNet. Many of the benchmarks in PIMbench are adapted from prior suites: GEMV (PrIM, InSituBench), GEMM (InSituBench, Phoenix), filter-by-key (PrIM, InSituBench, and bitweaving-style kernels in SIMDRAM), histogram (PrIM, IRAM, Phoenix), k-NN (InSituBench, SIMDRAM), VGG (SIMDRAM, DRISA), vector addition (PrIM), AXPY (InSituBench), radix sort (Pulley, a follow-on to InSituBench [44]), brightness (SIMDRAM), linear regression (Phoenix), and k-means (Phoenix). In addition, PIMbench introduces several applications (e.g., prefix sum, Page Rank, AES) that, to the best of our knowledge, have not appeared in prior PIM work.

3 DRAM Background

DDR (Double Data Rate), HBM (High Bandwidth Memory), LPDDR (Low Power Double Data Rate), and GDDR (Graphics Double Data Rate) are among the leading commodity DRAM technologies in modern computing, each with its own specific use cases.

DRAM Organization. DRAM is organized hierarchically, with individual memory bitcells grouped into progressively larger structures. At its core, DRAM consists of banks, subarrays, and MATs. Banks are independent units within DRAM that can perform operations in parallel. This allows for interleaving commands across banks, such that one bank can precharge while another is actively transferring data, improving throughput. Within each bank, memory is further divided into subarrays [35], which are smaller sections of memory designed to reduce electrical load and noise. Each subarray contains its own row decoders and local sense amplifiers, which together form the local row buffer. When a row in a subarray is activated, the contents of the row are loaded into the row buffer, enabling rapid access to that data. Successive accesses to the same row, known as row-buffer hits, can retrieve data directly from the row buffer without the need to close the row or activate a new one.

Subarrays themselves are composed of MATs, which are smaller, square arrays of memory cells, typically 512×512 in size. Each MAT contributes one or more bits to a memory read operation. However, in DDR, logically adjacent bits in a word of data (assuming traditional horizontal, row-oriented data layout) are scattered across multiple MATs within the subarray. This scattering complicates data access patterns and requires the memory controller to reassemble the bits in the global row buffer at the bank interface. Once the data is retrieved from the local row buffer, it is transferred via the global data lines (GDL) to the global row buffer at the bank interface and finally to the I/O pins. The width of the GDL

affects DRAM bandwidth, with DDR typically using a width of 64–128 bits, while HBM employs much wider GDLs, such as 256 bits. These hierarchical components: banks, subarrays, and MATs form the foundation of both DDR and HBM, though their organization differ between the two technologies.

DDR Organization. DDR (Figure 2a) is structured to balance cost and scalability, making it a popular choice for CPUs. A single DDR channel connects to one or more DIMMs (Dual Inline Memory Modules), with each DIMM consisting of one or more ranks. A rank is a collection of DRAM chips, and each chip contributes a subset of the data required for a memory access. For example, a rank containing eight chips would typically assign one byte of an eight-byte word to each chip. Within each chip, memory is divided into multiple banks, typically 16 or more per chip. To improve throughput, banks across multiple chips in a rank are grouped into logical banks. When the memory controller accesses a logical bank, it coordinates reads or writes from corresponding physical banks across all chips in the rank. This architecture scatters the bytes of a single memory word across multiple chips, introducing complexity for fine-grained operations, such as those required by PIM. Furthermore, all ranks and DIMMs on a DDR channel share the same address, data, and command bus. This shared-bus architecture limits parallelism, as only one rank or DIMM can be active at a time, resulting in contention. Additionally, DDR’s relatively narrow GDL further constrains bandwidth. DDR’s organization supports cost-effective and scalable memory access but introduces bottlenecks for applications that demand high bandwidth and parallelism. The scattering of bytes across multiple chips and the narrow GDL makes DDR less suitable for PIM [63], where precise control over data placement is crucial.

HBM Organization. HBM (Figure 2b), in contrast, adopts a different organizational hierarchy compared to DDR. HBM integrates multiple memory dies vertically into a single 3D-stacked package. These dies are interconnected using Through-Silicon Vias (TSVs). HBM eliminates the concept of ranks entirely; each HBM stack is divided into multiple independent channels, each with its own dedicated address, data, and command bus. This avoids bus sharing and contention, allowing all channels to operate simultaneously. Within the HBM stack, each channel directly accesses its assigned banks within the stacked memory dies. This direct channel-to-bank access eliminates the scattering of bytes across multiple chips seen in DDR, simplifying data placement and making HBM far more suitable for PIM. Furthermore, HBM’s banks are also divided into subarrays and MATs, like DDR. However, the combination of HBM’s significantly wider GDL (e.g., 256–512 bits) and independent channel architecture enables more parallelism for PIM. The reduced latency provided by its 3D-stacked architecture, HBM is suited for workloads that require high memory bandwidth and precise data placement, such as AI accelerators, GPUs, and PIM workloads.

LPDDR & GDDR. LPDDR was developed to deliver bandwidth comparable to DDR while significantly reducing power consumption. It retains the same core internal hierarchy—comprising banks, subarrays, and MATs—but introduces some architectural differences. Each LPDDR device is internally multi-channeled, similar to HBM, allowing for higher internal parallelism [48]. LPDDR also supports a wider GDL, typically 256 bits. GDDR, by contrast, is optimized for high-throughput, burst-oriented workloads in graphics and accelerator domains [48]. It shares a similar internal organization with DDR but incorporates wider (256–512 bit [14, 15]) GDLs, enabling higher read and write bandwidth. SK hynix’s AIM [37] demonstrates that lightweight compute units can be integrated into GDDR chips. Due to space constraints, this paper does not provide a detailed analysis of LPDDR- and GDDR-based PIM. However, both *PIMeval* and *PIMbench* have been extended to support LPDDR and GDDR.

4 PIM Variants

PIM can be categorized into Processing-in-Memory (PIM) and Processing-Near-Memory (PNM). PIM performs computation inside the DRAM chips themselves, while PNM places computation at the DRAM interface, such as on the

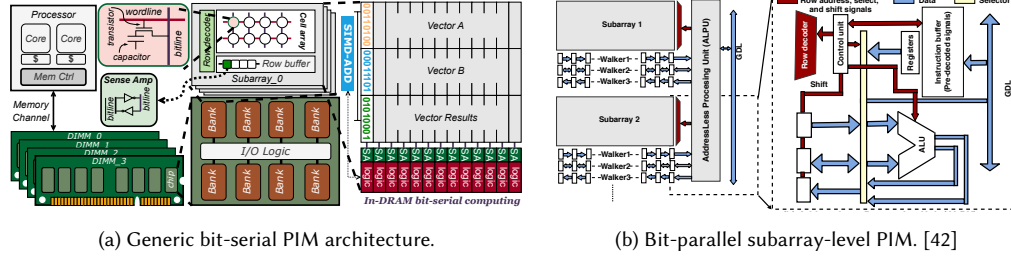


Fig. 3. PIM Variants

DIMM module (e.g., AX-DIMM [65]) or in the logic layer of a stacked DRAM such as HBM. Within the category of PIM, this paper compares three potential locations for integrating digital logic within DRAM, as shown in Figure 2a. Note that in all the techniques studied here, there is no bank-to-bank communication mechanism, so any computation requiring inter-bank coordination is offloaded to the host CPU. We assume the PIM program is executed by the memory controller, broadcasting each PIM operation to all subarrays/banks.

Subarray-level Bit-Serial. The first option, shown as (1) in Figure 2a, places the computation at the subarray level, with a simple bit processing element attached to each sense amplifier, with the bits of each data element laid out vertically up the bitlines. This allows a bit-serial computation on each bit in the entire subarray local row buffer to happen in unison, i.e. a bitslice computation. This architecture is shown in more detail in Figure 3a. This configuration offers subarray parallelism as well as row-wide bit-slice parallelism, and is sometimes also referred to as Processing-Using-Memory (PUM) [55]. The resulting bit-serial computation must perform at least n row accesses to operate on n -bit datatypes, and most compose more complex functionality (e.g., arithmetic) out of a few basic Boolean micro-operations, to minimize area overhead for the bit-level processing units attached to the sense amps, but makes up for this with the massive parallelism of operating simultaneously on the wide bit-slices. We model a *fully digital* subarray-level bit-serial PIM architecture inspired by previous works [20, 46].

Previous bit-serial PIM proposals, such as Ambit and SIMDRAM [27, 62], utilized analog computation based on charge sharing on the bitlines, implemented with triple row activation (TRA). TRA implements the MAJORITY function, but also requires dual-contact rows for NOT operations, which are costly [49]. TRA has the additional drawback that, to avoid large decoders for its functionality, only a small subset of rows support TRA, and operand rows must first be copied into these TRA rows. Due to process variation, aging, and area overheads for DCCs [49], DRAM vendors have expressed a preference for digital PIM approaches. DRISA [46] introduces a hybrid analog-digital model, but this shares the same concerns as a purely analog approach. In this paper, we model a digital bit-serial PIM architecture (shown in Figure 3a), modeled after Micron’s digital IMI architecture [20], with a few additional logic gates to implement associative (conditional match-update) processing, inspired by prior work on associative processing [8, 77]. This architecture, which we call DRAM-AP, features digital bit-serial logic connected to the sense amplifiers in each subarray, capable of performing XNOR, AND, and SEL (2:1 mux), supporting bit-serial arithmetic as well as associative processing. Additionally, it includes four extra bit registers to store intermediate values and condition bits, to achieve the conditional read and write needed for associative processing and the carry bit for arithmetic.

Subarray-level bit-serial has also been evaluated for accelerating a wide variety of applications, such as neural networks via early prediction and precision scaling [2] and random number generation [57]. Moreover, bit-serial arithmetic has prior precedent in digit-serial implementations of operators such as SAD [58]; Proteus [10] generalizes this principle to a wider range of workloads by introducing dynamic-precision PUM, demonstrating how adaptive bit-serial arithmetic can increase throughput in analog PUM. PIMSynth [24] presents a small-scale comparison of

fixed-precision digital and analog PIM. A thorough quantitative comparison within the vast design space of these analog and approximate bit-serial variants would be valuable; while PIMeval can be extended to support such a study, we leave this to future work.

Subarray-level Bit-Parallel. An alternative subarray-level PIM involves placing a more conventional bit-parallel, scalar ALU at the edge of the row buffer (Fulcrum [42], shown in Figure 3b). While Fulcrum was originally designed for 3D-stacked Hybrid Memory Cube (HMC) [32], we have adapted its design for DDR. The only change is that the original Fulcrum assumed that some operations could be offloaded to simple cores in the logic layer of the HMC, while our DDR version offloads computation to the CPU if it cannot be performed locally at the subarray. The essence of this architecture (Figure 3b) is a 32-bit, ALU shared between two consecutive subarrays. Each subarray pair is equipped with three Walker latches that capture input operands or store results before writeback, and an AddressLess Processing Unit (ALPU) comprising a controller, three temporary registers, an ALU, and an instruction buffer [42]. Read/write operations proceed sequentially via one-hot column selection. In this study, we model 32-bit ALPUs, so that a full 32-bit element is processed in one step.

Bank-level PIM. The third option, shown as (2) in Figure 2a, is to place a processing element at the bank interface, enabling bank parallelism, inspired by the BLIMP architecture [12]. BLIMP integrates a simple, in-order-issue 200 MHz RISC-V RV64GC processor without caches directly into each memory bank. BLIMP is designed such that banks operate independently, with each core associated with a single bank and able to access only the data within that bank. This enables all banks to operate in parallel, i.e., in all-bank mode. PIMeval’s bank-level model adopts the same assumption: banks operate independently and support all-bank computation, with the processing unit attached to each bank matching the width of the *global data line* (GDL) bus that brings data from the activated subarray to the bank interface; we model 128 for DDR. Data types smaller than the GDL width bits are processed in SIMD fashion (e.g., 4 32-bit ints per 128-bit GDL). While standard DDR configurations typically use a 64-bit GDL, our experiments use a 128-bit GDL based on the available DDR configuration from DRAMsim3 [13]. A GDL width sweep, presented in Section 10, shows that wider GDLs significantly benefit bank-level PIM throughput by enabling wider SIMD execution. Also note that, since the computation is placed at the bank interface, and before the bank-group-level muxing to the I/O interface, the bank-level parallelism is proportional to the number of banks (not bank groups).

PIMeval also incorporates a bank-level PIM model inspired by Samsung’s Aquabolt [41], allowing validation of simulation accuracy against the publicly available Aquabolt simulator [61]. The key distinction between BLIMP-style and Aquabolt-style PIM lies in both the underlying memory technology and execution model: Aquabolt targets HBM and assumes that only half of the banks are active during PIM operations, whereas BLIMP targets DDR and enables all-bank execution. We model a hybrid, with Aquabolt-style SIMD computation, but supporting a wider range of arithmetic operations and datatypes. Integrating a shared buffer, as in [30], is left for future work.

5 PIMeval

5.1 Overview

PIMeval is an extensible simulator implemented as a C++ library, designed to model PIM with vertical or horizontal data layouts. It features built-in performance and energy models for four PIM architectures (Section 4), and can be linked with benchmark programs to produce simulation executable. PIMeval adopts an analytical modeling approach, enabling fast and scalable evaluation while capturing key architectural behaviors and performance bottlenecks. It can also be integrated with cycle-accurate DRAM simulators [36, 47] via generated memory traces. The processing unit

is abstracted as a *PIM core*, representing a processing unit attached to either a subarray or a bank. PIMeval exposes a comprehensive set of high- and low-level APIs (Section 5.2) supporting device creation, resource management, data transfer, and a wide array of PIM operations, allowing the same benchmark code to target different PIM architectures. A resource manager handles data object allocation and tracking, where each data object can span multiple 2D memory regions across PIM cores to leverage subarray- or bank-level parallelism.

5.2 PIM API

PIMeval provides a high-level API set for modeling functional behavior, performance, and energy across multiple PIM architectures. This unified API set enables the same benchmark implementation to be reused across different architectures. It includes SIMD-style logical, arithmetic, and comparison operations, as well as broadcast, popcount, and reduction primitives. The API supports various integer types and Booleans, operating on PIM data object IDs, with the assumption that a software runtime manages resource allocation and address mapping. The API is designed to be extensible, allowing easy integration of new operations and data types. An example program, *AXPY*, implemented using the PIM API set, is shown in Listing 1.

```

1 void axpy(uint64_t vectorLength, const vector<int> &X, const vector<int> &Y, int A) {
2     // Allocate device memory
3     PimObjId objX = pimAlloc(PIM_ALLOC_AUTO, vectorLength, PIM_INT32);
4     PimObjId objY = pimAllocAssociated(objX, PIM_INT32);
5     assert((objX != -1) && (objY != -1));
6     // Copy inputs, perform operations, copy back results
7     pimCopyHostToDevice(X.data(), objX);
8     pimCopyHostToDevice(Y.data(), objY);
9     pimScaledAdd(objX, objY, objY, A)
10    pimCopyDeviceToHost(objY, Y.data());
11    // Free allocated memory
12    pimFree(objX);
13    pimFree(objY);
14 }
```

Listing 1. AXPY implementation using PIMeval API set.

5.3 Runtime Modeling

Runtime modeling is divided into two components: (i) data movement latency and (ii) kernel execution latency.

i. Data Movement: This is modeled based on the number of bytes transferred and the bandwidth of a memory rank, assuming all ranks operate as independent channels. For data movement within PIM across the DRAM hierarchy, we model two cases. Inter-subarray (within a bank) data movement is modeled using LISA [9], with movement overhead being $C_1 \times t_{RAS}$ if the row is open, or $(1 + C_1) \times t_{RAS}$ when row is closed. Here C_1 is a constant derived for LISA with default $C_1 = 0.143$. Inter-bank and inter-rank movement adopts the NDPBridge [70] abstraction which has two tier bridge. All the bank-level processing units are connected to the tier1 bridge (located at the buffer chip) that handles all cross-bank data movements within a rank via gather and scatter operations. Similarly, all the tier1 bridges are connected to the tier2 bridge (implemented as a software runtime in host CPU) that handles cross-rank data movements. For tier1, we model transfer time as the sum of a fixed per-transfer latency (64B granularity): $T_{\text{xfer}}(\text{bank} \rightarrow \text{bank})_{\text{intra-rank}} = T_0 + 2 \times \frac{S}{C_2}$. Here, S is the amount of data moved, C_2 is the intra-rank transfer rate, and the factor of 2 accounts for both the source readout and destination write-in. For tier2 movement, we add the cost of traversing the inter-rank link: $T_{\text{xfer}}(\text{bank} \rightarrow \text{bank})_{\text{inter-rank}} = T_{\text{xfer}}(\text{bank} \rightarrow \text{bank})_{\text{intra-rank}} + 2 \times \frac{S}{C_3}$, where C_3 is the inter-rank transfer rate. Note that, inter-PIM data movement can be realized through a variety of mechanisms

[66, 68, 84], ranging from host-mediated broadcasting to dedicated inter-rank communication fabrics. For instance, NDPBridge [70] supports low-latency intra-chip communication via a dedicated buffer at the cost of higher area overhead, whereas ABC-DIMM [68] provides no native intra-chip communication, requiring transfers to traverse the DIMM buffer. Such trade-offs underscore the need for flexible, parameterized modeling. Accordingly, this component has been parameterized in our simulator, allowing users to sweep communication latency to model different communication.

ii. Kernel Execution: Each PIM benchmark may consists of multiple PIM operations; either logical or arithmetic, with some benchmarks including kernels executed on the host, either due to random access, or expensive inter-bank communication. The latency of the host execution is measured directly using high-resolution clock. PIMeval models the execution time of each PIM operation based on the PIM device and DRAM parameters obtained from micron data sheet. The execution time of subarray-level bit-serial PIM is determined by factors such as operation type (e.g., arithmetic, logical) and data type (e.g., int32, bool). Each operation is implemented as a bit-serial microprogram composed of bit-serial micro-ops, to perform SIMD operations over bit-slices of input vectors with a vertical data layout. To accurately model the performance of the targeted bit-serial PIM, all high-level PIM APIs are mapped to low-level bit-serial microprograms, with manual tuning and optimization applied to improve performance. The runtime of subarray-level bit-parallel PIM is modeled by including row read/write latency to the local row buffer, as well as the latency required for subsequent arithmetic and logical operations. Bank-level bit-parallel PIM has been modeled by including row read/write latency to the global row buffer (in column size corresponding to the GDL width) and the latency to execute a PIM command by the processing unit. Besides SIMD PIM operations, some non-SIMD operations require special handling when modeling performance:

- (1) Reduction: Depending on PIM architectures, reduction can be performed with subarray or bank-level accumulators, or by the host CPU. Subarray-level bit-serial PIM can also perform row-wide pop counts for integer reduction sums, provided that appropriate hardware support is available.
- (2) Broadcast: This involves copying a scalar value to all elements of PIM objects. Bit-serial PIM can efficiently broadcast a bit value to an entire memory row, while bit-parallel PIM can propagate the value to a walker/register per subarray/bank.

5.4 Energy Modeling

Energy modeling consists of (i) data transfer energy, (ii) application execution energy, and (iii) background energy. PIMeval's approach is primarily based on the Micron power model [51].

i. Data Transfer Energy. The energy for data transfers between the PIM and the host, in both directions, is calculated using the Micron power model.

ii. Application Execution Energy. Application execution energy is modeled at the granularity of the PIM API calls. We aggregate the energy consumed by all the PIM APIs used in the application, depending on the specific PIM device that run the application. For instance, when *PIMAdd* is executed on a subarray-level PIM, it involves energy for row activation, precharging, data movement between the local sense amplifier, and ALU operations. However, when executed on a bank-level PIM, the energy associated with GDL transfers is also included. The energy for simultaneously activating or precharging multiple subarrays within a bank is derived using Equation 1. The GDL energy for transferring data within a bank, is scaled based on data from LISA [9]. To account for the ALU operation energy, we use values derived from RTL models for bit-serial and Fulcrum architectures, with the latter provided by the Fulcrum authors. We assume that the bank-level processing element consumes similar power as the Fulcrum ALU.

$$AP = V_{DD} \times (I_{DD0} \times (t_{RAS} + t_{RP}) - (I_{DD3N} \times t_{RAS} + I_{DD2N} \times t_{RP})) \quad (1)$$

Energy for applications that have host execution, is modeled by multiplying the host execution runtime with the TDP listed in table 2. Using TDP is pessimistic; incorporating actual CPU energy for the host portion is left for future work.

iii. Background Energy. The Micron power model does not account for the background energy when multiple subarrays or banks are active simultaneously. To address this, we model the background power of a single subarray by subtracting the standby power when the device is precharged from the standby power when the device is active. We then multiply this power by the total number of subarrays to determine the overall background power, which is further multiplied by the kernel execution time to calculate the background energy. For CPU idle energy, while it is waiting for a PIM operation to complete, we assume the cores are idle. Idle energy varies widely according to CPU model, but we use 10 W as a representative number. We performed a sensitivity analysis on this value, with minimal impact on most applications. For example, in vector addition, the PIM energy consumption for bit-serial architecture is 13.26 mJ, while the CPU idle energy is only 0.14 mJ, accounting for just 1% of the total energy. However, in applications with longer PIM execution times, such as VGG, the CPU idle energy increases correspondingly. For the same input size, VGG consumes 45k mJ of PIM execution energy and 22k mJ of CPU idle energy, leading to a 48% increase in total energy consumption.

5.5 Verification/Validation

We validate PIMeval along two dimensions: functional correctness and timing model. For functional verification, we compare the output of each benchmark against reference CPU implementation and require them to match exactly.

For performance validation, we validate PIMeval against two PIM simulators. First, For the evaluated configurations, the simulator achieves the identical performance as the Fulcrum simulator for Vector Add and AXPY. For GEMV and GEMM, PIMeval’s results are approximately 10% slower, which we attribute to overheads associated with PIMeval’s data-allocation mechanism. Second, we validate PIMeval by comparing against the cycle-accurate Aquabolt simulator [41] using identical DRAM timing parameters and architectural configurations. For vector-add workloads with input sizes ranging from 524,288 to 8,388,608 16-bit elements, the deviation ranges from -3.02% to $+8.21\%$ (avg. $+3.78\%$). For GEMV workloads with matrix sizes $131,072 \times 1,024$, $131,072 \times 2,048$, $262,144 \times 512$, and $262,144 \times 1,024$, the deviation is between $+14\%$ and $+18\%$ (avg. $+16\%$). We attribute the deviations primarily to Aquabolt’s ability to model and exploit overlap across banks (e.g., overlapping ACT/PRE and for even and odd banks), whereas PIMeval’s analytical model estimates the total number of ACT, PRE, and RD/WR operations and accounts for their costs in a serialized manner, except for overlap between the compute and RD/WR. The workloads here are selected because they are supported by the Fulcrum and Aquabolt simulators without additional implementation effort, enabling a direct comparison.

5.6 Configurability for Architectural Exploration

PIMeval supports configuring DRAM-related parameters (e.g., memory technology, timing) via configuration files similar to DRAMSim3 [47]. In contrast, architectural features require changes to the simulator code. For example, modifying the register file, processing-element, power models, or instruction set requires updating the PIM-specific performance and energy formulas. Functional simulation, on the other hand, does not require any code changes.

6 Methodology

Each PIM variant uses a DRAM configuration consisting of 10 ranks, each with 4 chips. Each chip contains 32 banks, and each bank has 16 subarrays arranged as 4096×8192 memory cell matrices. This configuration is chosen to provide the same memory capacity as an NVIDIA A100 GPU with 80 GB of HBM. For our baselines, we use the AMD EPYC 7742 64-core CPU and the NVIDIA A100 GPU [56]. Architectural parameters for PIM are summarized in Table 2. PIM

Table 2. Configuration of the Evaluated Architectures.

Architecture	Parameters
CPU	AMD EPYC 7742 64-core @ 2.3GHz, 225W TDP, 4MB L1, 64MB L2, 512MB L3, 1TB DDR4, 8 memory channels, memory BW 138.1GB/s.
GPU	NVIDIA A100-SXM4-80GB, 500W TDP, peak memory BW 2,280GB/s, peak compute throughput is 19.5 TFLOP/s.
Bit-serial	80GB DDR4, 10 ranks, 128 banks per rank, 16 subarrays per bank, and 8192-bit local row buffers. A bit-serial processing unit is attached to each sense amplifier in the local row buffer, supporting 4-bit registers and operations including move, set, and, xnor, and mux. Timing parameters: $t_{RAS} = 52$, $t_{RP} = 22$, $t_{CCD_S} = 4$, and $t_{CK} = 0.63$.
Fulcrum	80GB DDR4, 10 ranks, 128 banks per rank, 16 subarrays per bank, 8192-bit local row buffers. 32-bit 167 MHz ALU and three 8192-bit walkers are shared between every two subarrays. Timing parameters: $t_{RAS} = 52$, $t_{RP} = 22$, and $t_{CK} = 0.63$.
Bank-level	80GB DDR4, 10 ranks, 128 banks per rank, 16 subarrays per bank, 8192-bit local row buffers, 128-bit global row buffer, four 32-bit SIMD lanes, and three 8192-bit registers per bank. Timing parameters: $t_{RAS} = 52$, $t_{RP} = 22$, $t_{CCD_L} = 8$, and $t_{CK} = 0.63$.

configuration was chosen to ensure a fair comparison with the A100 GPU in terms of both memory capacity and total bank count [52]. DDR-based PIM has been used for the experiments; however, an analysis on DDR- vs. HBM-based PIM can be found in Section 10. It should also be noted that PIM performance scales almost linearly with number of ranks except for benchmarks such as BFS and PR, assuming the application datasets are large enough.

PIM is treated as a separate memory module, accessed via a dedicated memory controller or over CXL—analogous to a discrete accelerator. This separation is motivated by layout constraints in PIM. Bit-serial PIM, for instance, requires vertical bit layouts across rows [21], while bank-level and other PIM require densely packed, word-aligned data across banks/subarrays to maximize parallelism [12, 27]. Such layouts often conflict with standard address interleaving, necessitating explicit data movement and layout reorganization. Treating PIM as a separate memory module simplifies this process by enabling data to be explicitly copied and organized in a layout optimized for PIM operations. However, this choice introduces significant host-device and device-host data copy overhead (Figure 5). An alternative approach would involve a unified memory system [39], most likely with separate address mappings for PIM and host memory. Evaluating such a design would require system-level simulation, ideally using gem5, including mapping each PIM API (Section 5.2) to new m5 ops [6] and adding system calls to switch between PIM and host memory address spaces. It would also involve addressing execution semantics, e.g., whether the host should block during PIM execution, especially in multi-core systems where other cores may remain active. These complexities place such exploration beyond the current scope and are left for future work. Because PIM is treated as a separate memory module, we assume a bulk-synchronous execution model. Concretely, data required by a PIM kernel is copied from the host to PIM memory before invocation, and results are copied back before subsequent host computation (if needed). Accordingly, host execution, host↔PIM transfers, and PIM kernel execution are serialized at invocation boundaries (i.e., the host blocks during PIM phases), and there is no implicit coherence between host caches and PIM memory. In addition, invoking PIM as a separate module incurs a per-invocation launch/synchronization overhead, which we denote as T_{inv} . Thus, our end-to-end model is $T_{e2e} = T_{host} + T_{xfer} + T_{pim} + N_{inv} \cdot T_{inv}$, where N_{inv} is the number of PIM invocations (mostly one per benchmark in our evaluation). For a CXL.io-based Type-2 offload interface, we instantiate T_{inv} using published CXL offloading values [28] (e.g., $T_{inv} \approx 1.5 \mu s$ for a direct CXL.io launch), and expose T_{inv} as a configurable parameter.

PIMbench (Table 1) is implemented using our high-level PIM APIs (Section 7). GPU implementations leverage cuBLAS, Thrust, and CUB; triangle count, BFS, and PR use Gunrock [73]. CPU implementations use OpenMP, pthreads, OpenBLAS where applicable. GAP [5] benchmark is used for CPU graph implementation. VGG and ResNet use PyTorch. For fair comparisons, we assume PIM and GPU are connected to the host via PCIe or CXL and account for host–device data transfer in performance and energy analyses. CPU idle energy is excluded from PIM vs. GPU comparisons, as the host remains in a low-power state during accelerator execution.

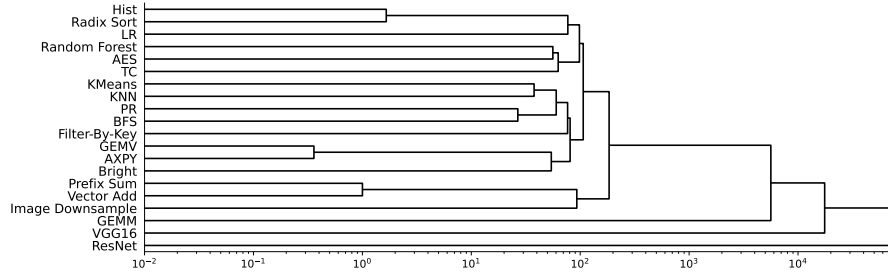


Fig. 4. Dendrogram showing similarity among benchmarks in PIMbench. The x-axis represents linkage distance on a logarithmic scale. A linkage distance ≤ 0 indicates that benchmarks are highly similar, differing only by minor variations in a single parameter.

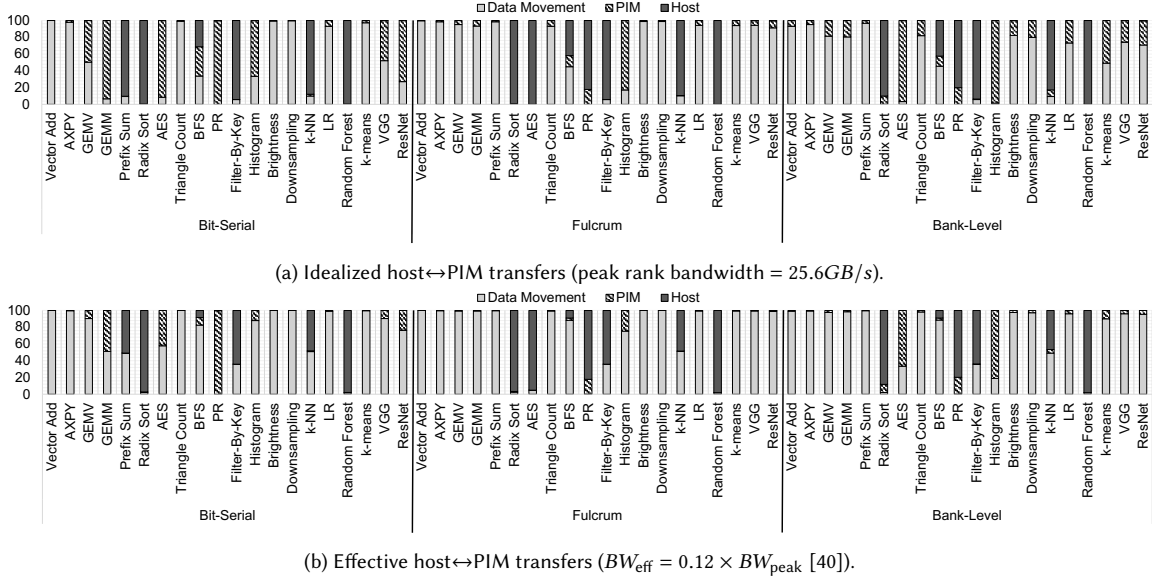


Fig. 5. Execution-time breakdown: percentage of total time spent on host↔PIM data movement, PIM execution, and host execution.

7 PIMbench: Analysis & Findings

This section presents the PIMbench suite, detailing the adaptations made for PIM architectures and analyzing the execution time and energy consumption of each benchmark across various PIM architectures, in comparison to baseline CPU and GPU. This work extends PIMbench suite [63] by incorporating deep learning model with residual connections (e.g., ResNet), control-intensive machine learning model (e.g., random forest), and data-parallel primitives (e.g., prefix sum). Table 1 lists the benchmarks, along with their input sizes, memory access patterns, and execution types. An execution type labeled as *PIM + Host* indicates that part of the benchmark is executed on the host processor, typically due to irregular or indirect access pattern. Figure 4 presents a dendrogram that quantifies benchmark diversity based on macro-op mix, memory access pattern, execution type, and arithmetic intensity. These features are first refined through Principal Component Analysis (PCA) and then clustered using hierarchical clustering [54] to generate the final dendrogram. The dendrogram reveals that benchmarks with similar macro-op mixes, execution types, and memory access patterns cluster together—e.g., AXPY and GEMV differ only in arithmetic intensity (0.17 vs. 0.5). Downsampling and Vector Add cluster at slightly higher linkage distance due to differing macro-op composition despite similar intensity (0.1 vs. 0.083) and access patterns.

Figure 5 shows the execution time breakdown for each benchmark, showing the percentage of time spent on data movement, host execution, and PIM kernel execution. The energy breakdown exhibits similar behavior and is not shown. For data movement, we report two end-to-end breakdowns: breakdown assuming peak aggregate rank bandwidth (Figure 5a) and breakdown using an effective transfer bandwidth (Figure 5b). We model effective bandwidth as $BW_{\text{eff}} = u \cdot BW_{\text{peak}}$ and scale transfer time as $T_{\text{eff}} = T_{\text{peak}}/u$, where u captures transfer inefficiencies (scheduling overhead and non-ideal memory level parallelization). Because u is system- and implementation-dependent, we treat it as a parameter; in Figure 5b we use the utilization factors reported by PIM-MMU [40] during DRAM $\leftrightarrow$ PIM copy operations as a representative conservative instantiation, and estimate transfer time using the bottleneck utilization (i.e., $u = \min(u_{\text{read}}, u_{\text{write}})$). Figure 6 shows the speedup achieved by each PIM variant across the benchmarks, compared to the CPU. This shows speedup considering both data movement and kernel execution time together, as well as speedup based solely on kernel execution time (which would be more relevant if the data are already resident in PIM memory and can be re-used). Figure 7 shows the speedup over the GPU; we neither include *cudaMemcpy* cost for GPU, nor *PimCopyHostToDevice* for PIM, to ensure fair comparison, as both can use PCIe/CXL. Since the PIM kernel invocation overhead is $\ll 0.2\%$ of the end-to-end execution time for all benchmarks and dataset of Table 1, we omit it from the figures. Figure 8 demonstrates the energy reductions relative to the CPU and GPU. The GPU comparison factors out CPU idle energy during GPU/PIM execution. In our prior work [63], bit-parallel PIM architectures did not receive a balanced workload distribution across PIM units. As a result, the sensitivity analysis showed limited performance improvement for Fulcrum and bank-level PIM as the number of ranks increased. In this paper, we address this limitation by allocating an equal amount of compute work to each PIM unit, resulting in improved execution time and scalable performance for bit-parallel PIMs. Due to space constraints, we omit detailed sensitivity analyses of PIM performance across varying rank, bank, and subarray counts. Interested readers are referred to our prior work [63].

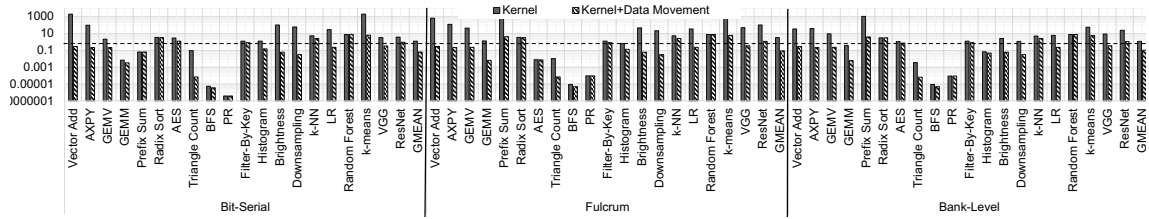


Fig. 6. Speedup over CPU. GMEAN speedups with and without data movement latency, respectively: Bit-Serial: 1.3, 0.07; Fulcrum: 3.2, 0.09; Bank-level: 1.2, 0.1.

Vector Addition. Vector addition is well-suited for execution on PIM architectures. Among the evaluated PIM variants, bit-serial PIM achieves the highest speedup compared to both CPU and GPU, followed by Fulcrum and bank-level PIM. A similar trend is observed in energy efficiency. In bit-serial, integer addition and boolean operations are $O(\text{bits})$ and fully exploit the bitslice parallelism of the subarray-level row buffer.

AXPY. AXPY is a linear algebra kernel that performs a scalar multiplication of one vector followed by an element-wise addition to another, involving both multiplication and addition operations. For this kernel, bit-serial and Fulcrum PIM architectures demonstrate similar performance, with Fulcrum performing slightly better, thanks to more efficient multiplication. In contrast, bank-level PIM is constrained by its narrow GDL width, which limits intra-PIM data throughput. As a result, both Fulcrum and bit-serial PIM achieve comparable speedups and energy reductions over CPU and GPU baselines, while bank-level PIM shows more modest improvements.

Matrix-vector Multiplication (GEMV). GEMV is a fundamental operation in linear algebra that involves matrix-vector multiplication. GEMV is implemented column-wise, scaling each column in the matrix and accumulating the

resulting partial-result columns (which is more efficient in PIM than performing row-wise dot products followed by a global sum-reduce) Due to its structure, Fulcrum and bank-level PIM architectures outperform bit-serial PIM in both performance and energy efficiency. Bit-serial PIM, in this case, fails to achieve a speedup over the GPU, whereas it can outperform CPU, as shown in Figures 6 and 7.

Matrix-matrix Multiplication (GEMM). We implemented GEMM using batched GEMV. Unlike GEMV, GEMM is compute-intensive, making it challenging for any PIM variant to efficiently execute GEMM, resulting in poor performance. However, Fulcrum demonstrates a speedup over the CPU if data movement latency is excluded. None of the PIM variants show energy savings compared to the CPU and GPU. Tiling strategies for PIM are left for future work.

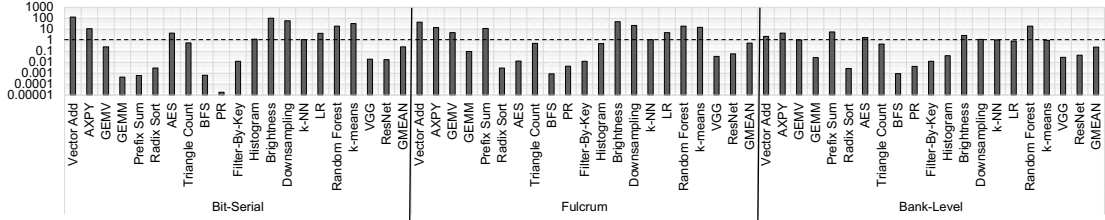


Fig. 7. Speedup Over GPU. GMEAN speedups: Bit-Serial: 0.5; Fulcrum: 0.5; Bank-level: 0.2.

Prefix Sum. Prefix sum is a data-parallel application that computes the cumulative sum of a vector, where each element in the output vector is the sum of all preceding elements in the input. Among the evaluated PIM architectures, Fulcrum and bank-level PIM execute prefix sum efficiently with assistance from the host to relay intermediate prefix sum values across subarrays/banks. In contrast, bit-serial PIM suffers due to the lack of inter-column communication. Overall, Fulcrum achieves the best performance, followed by bank-level PIM, while bit-serial is not a good fit.

Radix Sort. Radix sort iteratively groups elements into buckets based on the value of a specific digit position (radix). This digit-by-digit sorting approach allows radix sort to achieve favorable linear time complexity. For digit-by-digit sorting, we use counting sort, which involves both a counting and a sorting phase. The sorting phase requires data reshuffling, which is not supported in these PIM architectures, leading us to perform the counting phase in PIM, and the sorting phase on the host CPU. This approach introduces significant host latency (Figure 5), causing the PIM variants to show only slight speedup over the CPU and significant slowdown and energy consumption compared to the GPU.

Advanced Encryption Standard (AES). We implemented AES-256 in ECB mode [16], which operates on 16-byte state buffers over 14 rounds of logical and look-up operations. In our adoption, lookup table is implemented using a combination circuit [26], which makes it suitable for bit-serial processing. For bank-level PIM, the table is stored in registers. In contrast, Fulcrum lacks the ability to store the look-up table and offloads this step to the host. Among the three architectures, bit-serial achieves the highest performance due to its efficient support for logic operations and greater parallelism. Fulcrum performs the worst, as its reliance on host execution introduces significant overhead. For this distinct behavior, AES-ECB was chosen as benchmark for PIMbench suite.

Triangle Count. Triangle count determines the number of triangles (three mutually connected vertices) in a graph. To map the triangle count algorithm onto PIM, a series of AND, popcount, and reduction sum is employed [72]. Among the three PIM variants, bit-serial demonstrates the best performance, because AND is natively supported. However, the popcount and reduction steps are slower, with the net result showing only a slight speedup over the CPU and GPU for kernel-only. In contrast, both the Fulcrum and bank-level PIM variants fall short.

Breadth First Search (BFS). We implement FIFO-based BFS on PIM using CSR. Vertices are statically partitioned such that each core receives a contiguous range of vertices. The CSR row pointer and column index arrays are distributed in a

striped layout, with row pointers stored as core-local offsets. The frontier queue and visited array are maintained on the host. For each dequeued vertex v , the host broadcasts v to all PIM cores. Each core produces a one-hot ownership indicator. The host reads this to identify the owning core and v 's local slot within it. The local CSR offset for v is computed, and the host maps this offset to two striped positions corresponding to v and $v+1$ in the row pointer array. Two conditional copies extract the adjacency bounds from the PIM row index object. The host then constructs a neighbor mask over the striped column index layout for the range and uses a conditional copy to gather the corresponding neighbor IDs. The retrieved neighbor list is transferred to the host, where unvisited neighbors are marked and enqueued for the next frontier. None of the PIM variants outperform CPU or GPU (Figures 6 and 7), which is expected: PIM variants used in this study are not well-suited for irregular memory access and follow a bulk-synchronous execution model initiated at the memory controller level. As a result, only one PIM core processes a useful vertex at a time while the remaining perform redundant computation, limiting parallelism.

Page Rank (PR). We implement Page Rank using CSC (stored in host). PIM stores the out-degree array, rank vector, and nonzero-out-degree mask (precomputed once). Each PR iteration begins with calculation of dangling sum. For each vertex v , the host constructs a predecessor mask from the CSC structure, intersects it with the nonzero-out-degree mask on PIM, and conditionally gathers neighbor ranks and out-degrees. A PIM element-wise division and reduction yield v 's rank contribution, which the host combines with the dangling sum and damping factor to produce the updated rank. Convergence is checked on the host as the maximum absolute difference between old and new rank vectors. None of the PIM variants outperform CPU or GPU for PR (Figures 6 and 7), as expected: the bulk-synchronous execution model restricts rank updates to one vertex at a time, and each vertex incurs significant host-device communication overhead due to repeated neighbor mask transfers, outweighing any memory bandwidth benefit of PIM.

Filter-By-Key. Filter-By-Key scans a database table to select records matching a predicate. In our adaptation, filtering is performed on PIM, which generates a bitmap identifying 1% of matching records in a column. However, the host CPU must then fetch and process this bitmap to gather the selected items, making the gather phase the bottleneck (Figure 5). In contrast, CPU-only scans benefit from perfect spatial locality and prefetching. As a result, PIM variants do not outperform the GPU and show only modest gains over the CPU, with energy trends mirroring performance. Higher speedup is expected if more than one field per record were selected, as this would involve random accesses on CPU/GPU in a column-oriented layout.

Histogram. Histogram computes the distribution of RGB values in a 24-bit bitmap file. To mitigate the challenges posed by random memory access on PIM, we extract pixel values for each color channel (RGB) and sequentially traverse the key-value pairs (0–255). Using equality operations, we group instances of each value and apply a reduction to compute frequency counts. Among the PIM variants, bit-serial PIM demonstrates speedup and energy savings over both CPU and GPU. In contrast, Fulcrum and bank-level PIM struggle to achieve similar gains. This is primarily because our histogram implementation involves reduction operations, which can be efficiently implemented using popcount logic in bit-serial PIM but are less efficient on bit-parallel architectures like Fulcrum and bank-level PIM.

Brightness. Brightness, modeled after the SIMDram benchmark [27], increments each of the RGB values for every pixel of a 24-bit bitmap file by a set parameter. Given an input coefficient, each pixel of image data is processed with a saturating addition, ensuring the result stays within valid pixel value bounds through min and max operations. PIM achieves speedup and energy efficiency both over the CPU and the GPU.

Image Downsampling. PIMbench adapts box filtering for uncompressed bitmap images as an image downsampling candidate. Box filtering sets each pixel in the output image to the average of a box of input pixels. In our implementation, the image is scaled to half its size by applying addition and bit shifting operations. Since PIM can execute both of these

operations optimally, all three PIM variants outperform the CPU and GPU, while also demonstrating significant energy reductions.

k-NN. k-Nearest Neighbors (k-NN) is a supervised learning algorithm to classify individual points. Our PIM implemen-

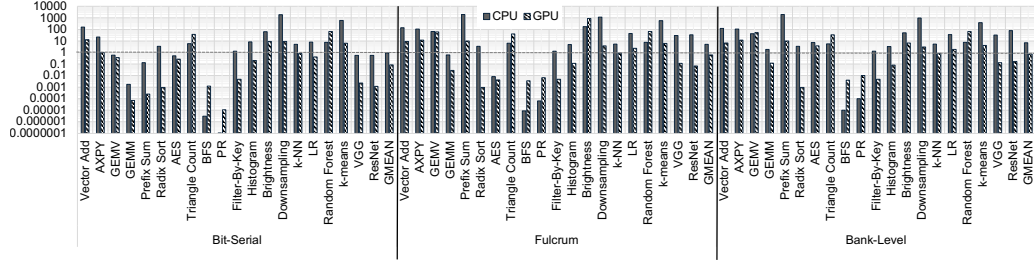


Fig. 8. Energy reduction.

tation adapts an end-to-end KNN batched inference where the sorting and classifying steps are executed on the host CPU, since PIM lacks support for shuffle. Distance computation, on the other hand, is executed in PIM. As shown in Figure 5 the sorting and classification phase contributes a significant percentage of the PIM execution latency. Despite this, we observed modest speedups and energy efficiency over baseline CPU and GPU.

Linear Regression. PIMbench implements a 2D variant of the linear regression. The method involves a single independent variable and is represented by the equation $y = \beta_0 + \beta_1 x + \epsilon$, where y is the dependent variable, x is the independent variable, β_0 is the intercept, β_1 is the slope, and ϵ is the error term. The slope and intercept are calculated using the least squares method. While bit-serial PIM is the most efficient for integer reduction, Fulcrum excels at multiplication. Considering that the ratio of reduction to multiplication is higher, Fulcrum and bit-serial exhibit similar performance, and all three PIM variants achieve speedup over both the CPU and GPU.

Random Forest. Random Forest [7] is a control-intensive machine learning model that consists of an ensemble of decision trees used for classification. Our implementation is inspired by prior work on adopting random forest for automata-based architectures [71]. In this design, the core inference logic, specifically, the tree traversal, is offloaded to PIM. For each input dimension, PIM evaluates all decision trees in parallel by performing logical comparisons between input features and model thresholds. These results are then aggregated across all dimensions using logical AND operations, ensuring only paths that satisfy all split conditions remain active. After all dimensions are processed, the resulting boolean vectors indicating active paths are copied back to the host. The host CPU then determines the final leaf node in each decision tree and performs a majority vote across all trees to compute the final classification result. This hybrid approach leverages PIM hardware to accelerate evaluation of complex tree ensembles while relying on the host for final aggregation. All three PIM architectures achieve speedup and energy savings vs. CPU and GPU.

k-means. k-means is a clustering algorithm that partitions a dataset into k distinct, non-overlapping clusters. Each data point is assigned to the nearest distance cluster, known as the cluster centroid. The algorithm iteratively refines the placement of centroids to minimize the within-cluster variance. The process involves assigning each data point to the nearest centroid and recalculating the centroids as the mean of all points with minimum distance to them. This iterative assignment phase entails a random access pattern, which is not well-suited for PIM. To address this, a bitmask is used to group data points belonging to each centroid. The mean of these grouped data points is calculated, updating the respective centroid. Given that our implementation uses simple PIM operations (e.g., subtract, add, equal), all three PIM variants show significant speedup and energy efficiency over both CPU and GPU.

CNN. VGG [64] is a Convolutional Neural Network (CNN) architecture composed of an input layer, multiple hidden

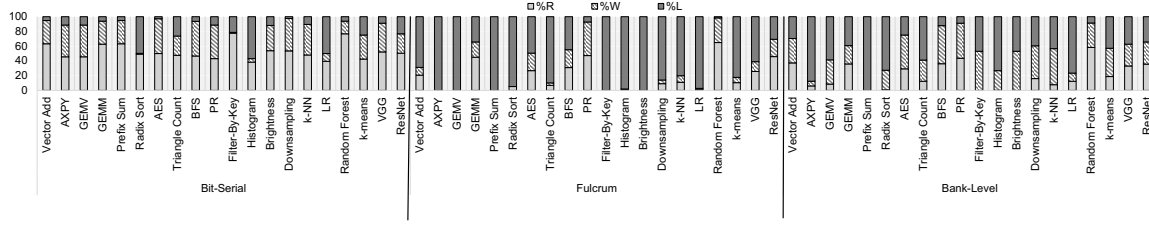


Fig. 9. Breakdown of PIM kernel execution time: Percentage spent on intra-PIM Read (%R), intra-PIM Write (%W), and Logic (%L).

layers (including convolutional, ReLU, max-pooling, and fully connected layers), and a softmax output layer. We implement three VGG variants: VGG-13, VGG-16, and VGG-19 – which differ in the number of convolutional layers and, consequently, in network depth. For space constraints, all results in this work use VGG-16. Running a full VGG network end-to-end on PIM is challenging due to preprocessing requirements such as padding at each layer. To address this, we decompose the VGG model into individual kernels corresponding to each layer. To exploit parallelism, we process input images in batches within PIM.

PIM architectures studied here support operations such as ReLU, max-pooling, and dense layers. However, the softmax layer includes exponentiation, which is not supported by any of these PIM variants and so must be offloaded to the host CPU. In addition, convolution layers involve strided access patterns, introducing further overhead. As a result, host-side processing and data movement (see Figure 5) create performance bottlenecks, limiting the speedup and energy efficiency gains of PIM over CPU execution. Nonetheless, PIM achieves moderate improvements across all VGG variants and PIM architectures. GPUs, however, still outperform PIM in both performance and energy efficiency.

We also implement ResNet [29], another widely used CNN architecture. Unlike VGG, ResNet incorporates residual connections that facilitate gradient flow and enable deeper networks without degradation. However, these connections introduce additional data dependencies and strided access patterns, increasing the complexity of PIM execution. As with VGG, we decompose ResNet into per-layer kernels. While PIM handles many individual layers efficiently, the added complexity from residual connections and convolutional strides limits speedup. As with VGG, PIM architectures offer modest gains over CPU execution but still fall short of GPU performance and energy efficiency.

8 Roofline Analysis

The roofline [75] provides a visual summary of the theoretical limit of an architecture with respect to memory bandwidth and peak performance. In addition to defining the theoretical performance ceilings of an architecture, roofline also exposes gaps between ideal and observed performance for applications. Benchmarks that fall below the architecture’s roofline indicate under-utilization, which may stem either from hardware constraints or algorithmic bottlenecks. Figure 10 presents a roofline model for five architectures: A100, EPYC 7742, bit-serial, Fulcrum, and bank-level PIM (Table 2). Each roofline for the corresponding architecture is constructed using Equation 2, which models the theoretical performance based on peak memory bandwidth and compute throughput as a function of operational intensity. Each roofline consists of two distinct regions. The slope represents the memory-bound region, where performance is limited by memory bandwidth. The horizontal plateau corresponds to the compute-bound region, where throughput is capped by the architecture’s maximum throughput. The point at which the roofline transitions from slope to plateau is referred to as the ridge point (or machine balance) indicating the operational intensity at which a workload becomes compute-bound on a given architecture. For the CPU and GPU, peak bandwidth and throughput are obtained from Hopschotch [1] and A100 datasheet [56], respectively. For PIM architectures, these values are computed using Equations 3 and 4 (In Equation 4, *Active PIM Compute Unit* denotes the number of subarrays/banks, and *Compute Frequency* denotes

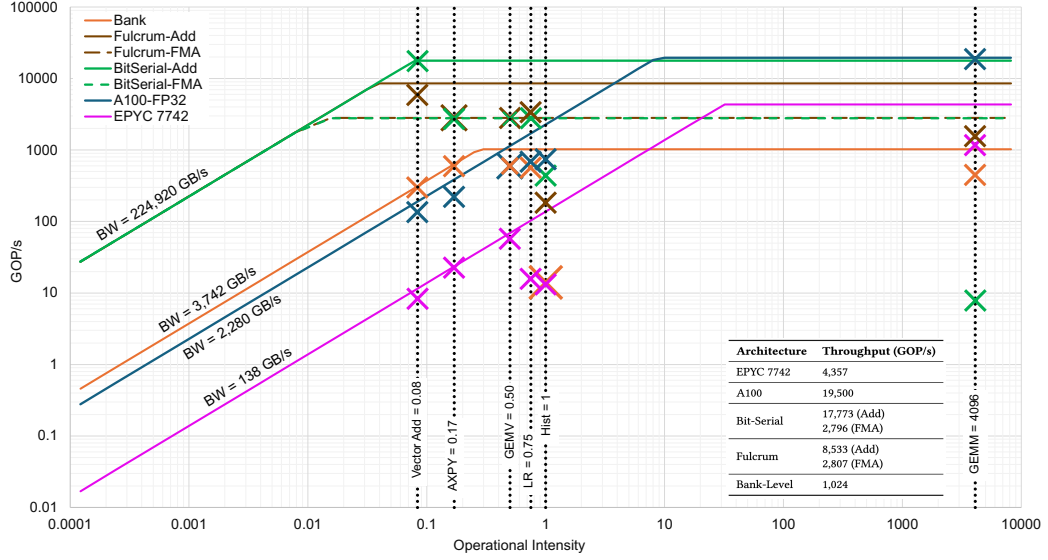


Fig. 10. Roofline model for five architectures: NVIDIA A100 (blue), AMD EPYC 7742 (pink), bit-serial (green), Fulcrum (brown), and bank-level (orange). Both bit-serial and Fulcrum are plotted assuming 8 active subarrays per bank, totaling 10K PIM cores. Bank-level PIM is configured with 128 banks per rank, totaling 1K PIM cores. The plot is shown on a log-log scale, with operational intensity (x-axis) ranging from 0.0001 to 8192. Vertical dotted lines denote the operational intensities of representative benchmarks. Color-coded cross marks indicate the achieved throughput for each architecture at the corresponding operational intensity. For instance, the pink cross on the vertical line labeled "Vector Add" represents the throughput achieved by the CPU for the vector addition benchmark.

the operating frequency of the PIM), based on parameter values provided in Table 2. To capture operation-specific variability in bit-serial and Fulcrum, we log throughput for add and fused multiply add (FMA).

$$y_i = \min(\text{Peak Performance}, \text{Peak BW} \times x_i) \quad (2)$$

$$\text{BW (GB/s)} = \frac{\text{\#Bits}}{8 \times (t_{\text{RAS}} + (t_{\text{CCD_L}} \times \text{itrgDL}) + t_{\text{RP}}) \times t_{\text{CK}}} \quad (3)$$

$$\text{Performance (GOP/s)} = \frac{\text{Active PIM Compute Unit}}{\text{Compute Frequency}} \quad (4)$$

Figure 10 also presents a roofline analysis of each architecture using a selection of benchmarks from the PIMbench suite (Table 1). Each benchmark is annotated with its corresponding operational intensity, represented by a vertical dotted line. The selected benchmarks span a range of operational intensities, highlighting different compute and memory characteristics. Benchmarks such as VGG, ResNet, and k-NN, which are composed of multiple building block kernels, exhibit wide variability in operational intensity. This makes it difficult to assign a single representative data point for these workloads. Analyzing these complex benchmarks by decomposing them into their constituent kernels would provide important insights and is left for future work.

Before delving into architecture-specific analysis with respect to the benchmarks, we highlight a few observations. Subarray- and bank-level PIM architectures can get up to orders-of-magnitude memory bandwidth improvements over high-end CPU/GPU: approximately $\sim 1630\times$ and $\sim 28\times$ higher than EPYC 7742, and $\sim 99\times$ and $\sim 2\times$ higher than A100, respectively. On the other hand, when comparing compute throughput, CPUs and GPUs still hold an edge. Subarray-level PIM performance scales with the number of subarrays, while bank-level performance scales with the GDL width as well as number of banks. The A100 reaches 19,500 GOP/s, significantly higher than the 4,357 GOP/s of the EPYC 7742 CPU. While bit-serial PIM achieves the highest throughput for addition (17,773 GOP/s), its performance

for fused multiply-add (FMA) operations (2,796 GOP/s) is comparable to that of the CPU. Fulcrum achieves 8,533 GOP/s for addition and 2,807 GOP/s for FMA, whereas bank-level PIM lags behind with 1,024 GOP/s.

While PIM benefits from massive memory bandwidth, CPU and GPU performance can catch up depending on the workload’s operational intensity (OI). For instance, GPU breaks even with bank-level PIM at $OI > 0.5$, and surpasses Fulcrum and bit-serial PIM for FMA at $OI > 1.5$. However, Fulcrum’s addition throughput holds out longer, requiring $OI > 4$ before GPU overtakes it. CPU, on the other hand, requires $OI > 7$ to match bank-level PIM and reaches parity with Fulcrum and bit-serial FMA only at $OI \approx 32$. Readers are advised to interpret these inflection points as illustrative rather than absolute, as they depend on specific CPU, GPU, or PIM architecture. For example, in the studied PIM architectures, varying the subarray, bank or rank count while keeping other parameters constant, leads to linear changes in both the roof (peak performance) and the slope (peak memory bandwidth), thereby shifting the inflection points.

We now delve into benchmark-based analysis to better understand how each platform performs across the selected workloads. It can be observed that EPYC achieves high performance on benchmarks such as Vector Add, AXPY, GEMV, and Linear Regression (LR), getting over 70% of its theoretical peak performance (Table 3). However, Histogram (Hist) exhibits significantly lower performance, achieving only 9.42%, primarily due to irregular memory access patterns and poor spatial locality. GEMM achieves only 34.72% of peak performance, despite utilizing a highly optimized BLAS library. In contrast, A100, across all benchmarks, consistently achieves performance close to its theoretical peak. Notably, it achieves 95.81% of peak performance on the GEMM. For benchmarks such as AXPY and GEMV (both implemented using cuBLAS), Nsight Compute (NCU) analysis indicates that performance could be improved through better load balancing. In the case of the Hist (implemented using CUB), NCU reports that the compute pipeline is underutilized due to uncoalesced memory accesses—leading to stalls and inefficient bandwidth usage—limiting performance.

By co-locating compute units with memory, PIM significantly reduces data movement, but its performance is constrained by limited compute capabilities, limited registers, and lack of inter-bank/inter-chip communication. Bit-serial and bank-level PIM architectures achieve near-peak performance on Vector Addition, reaching 98.46% and 99.75% of their respective ceilings. In contrast, Fulcrum achieves only 68.7%, due to architectural constraints – the use of only three walkers per subarray limits its ability to overlap intra-DRAM read/write with computation. As a result, the ALU remains idle for at least 30% of total execution time (Figure 9). All three PIM architectures achieve over 90% of their peak performance for AXPY. This is primarily due to the scalar value being stored in a register across all designs, allowing the compute units to remain fully utilized with minimal stalls from intra-DRAM read/write operations (Figure 9). For GEMV, both the bit-serial and Fulcrum architectures perform well, achieving over 95% of their peak performance. In contrast, bank-level PIM reaches only 58% of its peak throughput. As illustrated in Figure 9, this is because the compute unit in the bank-level PIM remains idle 40% of the time, waiting for data transfers between the DRAM rows and the bank interface. For LR, bank-level PIM achieves a reasonable level of efficiency, reaching approximately 55% of its peak performance. However, as shown in Figure 9, this performance could be improved further if read and write could be overlapped with logic execution. In contrast, Fulcrum struggles to surpass 39%, and bit-serial PIM achieves only 15%. LR has both a moderate amount of *multiplication*, plus *reduction* operations. As a result, while the achieved throughput exceeds the FMA roofline of both architectures, the dominance of multiplication keeps it only slightly above that bound. The Histogram benchmark uses reduction, with the host computing the final value, due to the lack of inter-subarray/bank communication. Each subarray or bank performs a local reduction, which is read by the host for final aggregation. Since this is included in PIM performance, none of the PIM architectures achieves near-peak performance. Despite being representative of compute-intensive workloads, GEMM performs poorly on all PIM architectures due to the lack

of on-chip buffers, which hinders data reuse and causes redundant intra-PIM read/writes. As a result, compute units remain idle 70% of the time (Figure 9).

Table 3. Achieved vs. ceiling performance, and achieved % across architectures.

Benchmark	OI	EPYC 7742			A100			BitSerial			Fulcrum			Bank-Level		
		Ach	Ceil	%	Ach	Ceil	%	Ach	Ceil	%	Ach	Ceil	%	Ach	Ceil	%
Vector Add	0.083	8	11	72.73	134	182	73.63	17499	17773	98.46	5861	8533	68.70	298	299	99.75
AXPY	0.17	22	24	91.67	220	388	56.70	2718	2796	97.22	2804	2807	99.93	594	636.13	93.44
GEMV	0.5	57	69	82.61	598	1140	52.46	2737	2796	97.91	2804	2807	99.93	598	1024	58.44
LR	0.75	94	104	91.06	685	1710	40.06	2784	17773	15.67	3367	8533	39.46	564	1024	55.08
Hist	1	13	138	9.42	740	2280	32.46	438.7	17773	2.47	182.72	8533	2.14	11	1024	1.07
GEMM	4096	1512	4357	34.72	18683	19500	95.81	3	2796	0.11	1560	2807	55.59	447	1024	43.63

Color legend: ■ ≥95%, ■ 85–94%, ■ 70–84%, ■ 55–69%, ■ 30–54%, ■ 11–29%, ■ ≤10%.

9 Optimization of Bank-Level PIM

PIM struggles to outperform high-end GPUs for high-operational-intensity workloads such as GEMM (Section 8). This outcome is expected: high operational intensity pushes GEMM toward the compute-bound regime, and PIM without sufficiently high compute throughput are unlikely to match GPU performance. In this section, we focus on bank-level PIM and quantify how GEMM performance varies with compute and execution mechanisms. One approach to improving throughput is to increase parallelism (e.g., activating more banks, or scaling across ranks/HBM stacks), but these gains are bounded by the area/power constraints. Performance for compute-intensive kernels also depends on how effectively the design keeps the compute units busy, i.e., minimizing stalls caused by redundant data movement and DRAM command overhead. We study two representative mechanisms. First, we evaluate a *software* optimization—instruction fusion—which combines sequences of operations (e.g., multiple MAC/MAD/ADD) to avoid redundant row reads/writes and better utilize the available compute resources. Relative to our baseline bank-level implementation, Fusion reduces GEMM kernel time by 2.57×. Second, we evaluate a *bank-level execution* mechanism that exploits subarray-level parallelism [35] to overlap DRAM row operations within a bank (e.g., overlapping ACTIVATE/PRECHARGE across subarrays). This improves compute utilization by reducing compute idle time due to serialized row-management events. With this optimization (SALP), GEMM kernel time improves by 2.75×.

10 Impact of DRAM Technologies on PIM

All the results presented thus far have focused on DDR-based PIM, PIMeval and PIMbench support all the DRAM variants. This section provides a summary of how the underlying DRAM standard — specifically DDR versus HBM — impacts PIM performance. We omit results for LPDDR and GDDR due to space constraints, but LPDDR and GDDR will show higher bank-level throughput than DDR, due to their wider GDL (256/512 bits).

From a system bandwidth perspective, HBM significantly outperforms DDR, offering memory bandwidths in the hundreds of GB/s, compared to tens of GB/s for DDR. In our evaluation, the DDR-based system sustained 25.6GB/s for one rank, while a single HBM stack delivered 285GB/s. As shown in Figure 5, data movement latency between the host and PIM dominates the overall kernel execution time. Consequently, switching to HBM-based PIM reduces execution time across all PIM variants by approximately 11×, when host–PIM data transfer is considered. However, benchmarks that require host execution (Table 1; Figure 5; Section 7) do not benefit from this bandwidth improvement, as the host portion dominates total runtime. Following Amdahl’s Law [3], the overall speedup in such cases remains negligible.

Both DDR and HBM rely on the traditional sequence of DRAM commands: *ACTIVATE*, *READ/WRITE*, and *PRECHARGE*, with timing variations depending on the generation. However, our analysis shows that the timing parameters for DRAM commands (e.g., t_{RAS} , t_{RP}) are comparable between the DDR and HBM. Consequently, subarray-level PIM exhibits negligible performance difference *if and only if* the number of subarrays per bank is held constant. However, HBM has a lower #rows per bank compared to DDR, so maintaining the same #subarrays requires fewer #rows per subarray. Conversely, if the #rows per subarray is kept constant, the number of subarrays must be reduced, leading to a linear increase in execution time. In contrast, bank-level PIM is influenced by the underlying differences in GDL width. As shown in Figure 11a, HBM-based bank-level PIM achieves nearly 2× the performance of its DDR-based counterpart across all benchmarks from the PIMbench suite, despite having the same number of banks and total memory capacity (128 GB). This improvement stems entirely from HBM’s support for a wider SIMD datapath, enabled by its 256-bit GDL, compared to DDR’s 128-bit GDL. However, benchmarks such as radix sort and filter-by-key delegate portions of their execution to the host processor. As a result, host-side execution dominates overall runtime (as shown in Figure 5), and per Amdahl’s Law [3], these benchmarks exhibit little to no visible performance improvement. To further isolate the effect of GDL width on bank-level PIM, we varied GDL width while keeping memory capacity and bank count fixed. As shown in Figure 11b, performance improves linearly, reinforcing the impact of GDL width on bank-level PIM throughput. Graph analytics benchmarks (BFS and PageRank) exhibit negligible sensitivity to DRAM technology. The memory-controller-level bulk execution model in PIM leads to poor load balancing shifting the bottleneck to execution serialization rather than memory technology specific parameters.

11 Architectural Insight

Bit-serial PIM is effective for highly parallel, bitwise kernels. It benefits applications dominated by element-wise operations, such as vector addition, AXPY, AES, and image-processing benchmarks consisting of additions and logical operations. Unlike conventional DRAM, where row activation is amortized over a burst of reads/writes, bit-serial PIM must activate and precharge a row for every bit operation, making overall performance fundamentally bounded by the DRAM row activate and precharge latency (t_{RAS} and t_{RP}). As a result, its advantages diminish for small datasets where rows are only partially utilized and activation overhead dominates. Bit-serial PIM is also ill-suited for kernels requiring inter-element communication, such as Prefix Sum, where the i -th output depends on all preceding elements. Since each element occupies a separate bitline and no inter-bitline communication mechanism exists, such dependencies must be resolved by the host. Finally, relative to the other PIM variants studied here, bit-serial PIM underperforms on multiplication- and division-heavy benchmarks such as GEMV, GEMM, CNNs, and Page Rank because the cost of multiplication/division grows quadratically with operand bit-width.

Subarray-parallel bit-parallel PIM is sensitive to subarray-level parallelism. Its performance scales roughly linearly with the number of subarrays that can be activated in parallel. For applications such as vector addition, subarray-parallel bit-parallel PIM does not exhibit the massive speedups that subarray-level bit-serial PIM can achieve under ideal row utilization, because each element is still processed sequentially within a subarray and overall parallelism is bounded by the number of active subarrays. That said, subarray-level bit-parallel PIM clearly outperforms the other PIM variants studied here for applications such as GEMV, GEMM, and CNNs. For data-parallel applications like Prefix Sum, it still cannot communicate directly between subarrays due to the lack of inter-subarray communication; however, the performance and energy impact is less severe than in the bit-serial design, because the host can pipeline reads and writes from multiple subarrays in parallel and relay partial sums across them.

Bank-level PIM is sensitive to GDL width. It uses SIMD unit to process an entire GDL-width chunk of data every

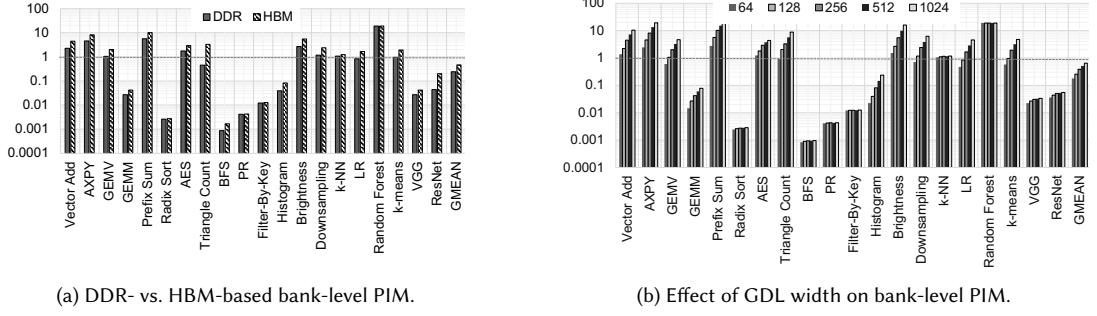


Fig. 11. Runtime analysis of DRAM technology, normalized to NVIDIA A100. Bank-level PIM benefits from wider GDL.

t_{CCD_L} interval, and can, in principle, outperform subarray-level bit-parallel PIM when the GDL width is sufficiently large. For example, with our configuration, the effective parallelism for subarray-level bit-parallel PIM is on the order of $10 \times 128 \times 8 \approx 10K$ elements per PE cycle, whereas bank-level PIM can reach $10 \times 128 \times 16 \approx 20K$ elements per t_{CCD_L} when the GDL width is 512 bits and t_{CCD_L} is smaller in cycles than the PE latency. However, in the DDR-based bank-level configuration used here (with a narrower GDL and conservative timing parameters), bank-level PIM does not outperform subarray-level bit-parallel PIM for any of the applications we study, although it does outperform bit-serial PIM for multiplication-heavy workloads such as GEMV, GEMM, CNNs, and for data-parallel kernels such as Prefix Sum.

Memory-controller-level bulk execution limits parallelism for graph or any workloads requiring indirect/random access. Our BFS and PageRank results reveal an important distinction between memory-controller-level bulk execution, as used in the PIM variants studied here, and per-core execution as in UPMEM. Because the execution model initiates operations across all cores simultaneously from the memory controller, only one core performs useful work per step while the remaining cores execute redundant operation. For BFS, this means one vertex is processed at a time; for PageRank, rank updates are similarly serialized. This fundamentally limits parallelism for graph workloads, where frontier sizes and neighbor counts are irregular and dynamic, and suggests that per-core execution models are better suited for such applications.

12 Limitations

Simulator limitations. PIMEval does not yet enable comprehensive design-space exploration for PIM in several ways. First, it does not support pipelined execution (e.g., as in UPMEM [19], which is now discontinued and no longer accessible to the community). Second, it does not incorporate cycle-accurate performance and energy modeling. Third, certain PIM-specific microarchitectural parameters (e.g., register file size, functional-unit configuration) require code modification. Fourth, the analytical-model-based simulator assumes a simplified system integration model (PIM-host data movement, memory and page allocation). Fifth, the current framework supports only the bulk-synchronous execution model, limiting exploration of MIMD-PIMs [59] and support for irregular applications such as BFS and PageRank, which require divergent control flow and non-uniform memory accesses across PIM cores. Finally, the framework has not been validated against real-world PIM hardware, though it has been validated against prior simulators [42, 61]. As a result, absolute accuracy may be limited, although relative performance and energy trends remain informative.

Characterization limitations. This paper also has several limitations in terms of PIM characterization. The PIM design space is inherently multidimensional, spanning DRAM technology, DRAM organization, PIM microarchitectural choices, and software optimizations such as instruction fusion. Because the framework currently lacks a compiler, many of

these parameters cannot be explored dynamically, limiting our ability to study how changes along different dimensions interact. Furthermore, we do not perform area-level characterization (e.g., performance per unit area).

13 Conclusions, Discussion, and Future Work

This paper extends *PIMbench* and *PIMeval* [63] to further analyze tradeoffs among three representative PIM designs at different levels of memory integration: subarray-level bit-serial (DRAM-AP), subarray-level bit-parallel (Fulcrum), and bank-level PIM. Through roofline analysis and execution time breakdowns, we identify key performance bottlenecks such as ALU underutilization and intra-memory data movement. We further study the impact of DRAM technology, demonstrating that while subarray-level PIM is relatively insensitive to DRAM variation, all PIM designs benefit from HBM’s increased bandwidth. Notably, bank-level PIM achieves up to a 2× speedup when using HBM, attributed to its wider GDL, which enable greater parallelism. Our analysis underscores that *no single architecture—CPU, GPU, or PIM—consistently outperforms the others across all workloads*. As expected, GPUs excel on compute-intensive workloads, such as those involving matrix multiplication (e.g., CNNs), while PIM outperforms both CPU and GPU on memory-bound benchmarks such as GEMV, Random Forest, and Prefix Sum. Although the Gmean speedup of PIM vs. CPU and GPU is modest, this masks dramatic variation among applications. Roofline modeling highlights the disparity in memory bandwidth and compute throughput across PIM variants, as well as among PIM, CPU, and GPU architectures, and across benchmarks. Roofline analysis also shows crossover points in OI between PIM vs. CPU/GPU computation. For example, with 10 ranks of bank-level PIM, the GPU will be preferable from a performance standpoint for kernels with an OI greater than 0.5 (although the crossover point for energy efficiency may be considerably different). While this study focuses on three specific PIM designs, we acknowledge a broader design space that includes bank-group, rank-level, or memory controller-level integration, each with distinct trade-offs. Future work will explore additional architectural configurations, system integration strategies, and compiler support.

Acknowledgment

This work was supported in part by PRISM, one of seven centers in JUMP 2.0, an SRC program sponsored by DARPA; the NSF under grant PPOSS-2217071 and the NSF I/UCRC MIST Center, grants IIP-1439644, IIP-1439680, IIP-1738752, IIP-1939009, IIP-1939050, and IIP-1939012; Booz Allen Hamilton under contract FA-8075-18-D-0004; and SRC contract 2019-NM-2875. We thank Marzieh Lenjani for Fulcrum-related pointers and the anonymous reviewers for feedback. AI tools were used for text polishing.

References

- [1] A. Ahmed and K. Skadron. 2019. Hopsotch: a micro-benchmark suite for memory performance evaluation. In *MemSYS*. 167–172.
- [2] V. Akhlaghi et al. 2018. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *ISCA*. IEEE, 662–673.
- [3] G M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. 483–485.
- [4] T. Andrulis, J. S. Emer, and V. Sze. 2024. CiMLoop: A Flexible, Accurate, and Fast Compute-In-Memory Modeling Tool. *arXiv:2405.07259* (2024).
- [5] S. Beamer, K. Asanovic, and D. A. Patterson. Aug. 2015. The GAP benchmark suite. *arXiv arXiv:1508.03619* (Aug. 2015).
- [6] N. Binkert et al. 2011. The gem5 simulator. *ACM Special Interest Group on Computer Architecture (SIGARCH) Computer Architecture News* 39, 2 (May 2011), 1–7.
- [7] Leo Breiman. 2001. Random forests. *Machine learning* 45 (2001), 5–32.
- [8] H. Caminal and other. 2021. CAPE: A content-addressable processing engine. In *HPCA*. 557–569.
- [9] K. K. Chang et al. 2016. Low-Cost Inter-Linked Subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. In *HPCA*. 568–580. <https://doi.org/10.1109/HPCA.2016.7446095>

- [10] G. F. De Oliveira Junior et al. 2025. Proteus: Achieving High-Performance Processing-Using-DRAM with Dynamic Bit-Precision, Adaptive Data Representation, and Flexible Arithmetic. In *Proceedings of the 39th ACM International Conference on Supercomputing*. 473–494.
- [11] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (1974), 256–268. <https://doi.org/10.1109/JSSC.1974.1050511>
- [12] A. Devic, S. B. Rai, A. Sivasubramaniam, A. Akel, S. Eilert, and J. Eno. 2022. To PIM or not for emerging general purpose processing in DDR memory systems. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 231–244.
- [13] DRAMsim3. 2025. DDR4-8Gb-x16-3200. https://github.com/umd-memsys/DRAMsim3/blob/master/configs/DDR4_8Gb_x16_3200.ini.
- [14] DRAMsim3. 2025. GDDR5-1Gb-x32. https://github.com/umd-memsys/DRAMsim3/blob/master/configs/GDDR5_1Gb_x32.ini.
- [15] DRAMsim3. 2025. GDDR5X-8Gb-x32. https://github.com/umd-memsys/DRAMsim3/blob/master/configs/GDDR5X_8Gb_x32.ini.
- [16] M. J. Dworkin. 2001. Recommendation for block cipher modes of operation: Methods and techniques. In *NIST Special Publication 800-38A*.
- [17] D. G. Elliott et al. 1999. Computational RAM: Implementing processors in memory. *IEEE Design & Test of Computers* 16, 1 (1999), 32–41.
- [18] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*. 365–376.
- [19] Yann Falevoz and Julien Legriel. 2023. Energy efficiency impact of processing in memory: A comprehensive review of workloads on the upmem architecture. In *European Conference on Parallel Processing*. Springer, 155–166.
- [20] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning. 2017. In-memory intelligence. *IEEE Micro* 37, 4 (2017), 30–38.
- [21] B. R. Gaeke, P. Husbands, X. S. Li, L. Oliker, K. A. Yelick, and R. Biswas. 2002. Memory-intensive benchmarks: IRAM vs. cache-based machines. In *IPDPS*. 7 pp. <https://doi.org/10.1109/IPDPS.2002.1015506>
- [22] S Ghose et al. 2019. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development* 63, 6 (2019), 3–1.
- [23] Juan Gómez-Luna et al. 2023. Evaluating machine learning workloads on memory-centric computing systems. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 35–49.
- [24] D Guo et al. 2025. PIMsynth: A Unified Compiler Framework for Bit-Serial Processing-In-Memory Architectures. *CAL* (2025).
- [25] J. Gómez-Luna et al. 2021. Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-In-Memory Hardware. In *Proceedings of the International Green and Sustainable Computing Conference (IGSC)*. 1–7. <https://doi.org/10.1109/IGSC54211.2021.9651614>
- [26] Omid Hajihassani, Saleh Khalaj Monfared, Seyed Hossein Khasteh, and Saeid Gorgin. 2019. Fast AES implementation: A high-throughput bitsliced approach. *IEEE Transactions on parallel and distributed systems* 30, 10 (2019), 2211–2222.
- [27] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. D. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu. 2021. SIMDram: a framework for bit-serial SIMD processing using DRAM. In *ASPLOS*. 329–345.
- [28] H Ham et al. 2024. Low-overhead general-purpose near-data processing in cxl memory expanders. In *MICRO*. IEEE, 594–611.
- [29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
- [30] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. N. Vijaykumar. 2020. Newton: A DRAM-maker's accelerator-in-memory (AiM) architecture for machine learning. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 372–385.
- [31] John L Hennessy and David A Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60.
- [32] Hybrid Memory Cube Consortium (HMCC). 2015. *Hybrid Memory Cube Specification 2.1*. <https://www.hybridmemorycube.org/>.
- [33] Bongjoon Hyun, Taehun Kim, Dongjae Lee, and Minsoo Rhu. 2024. Pathfinding future pim architectures by demystifying a commercial pim technology. In *HPCA*. IEEE, 263–279.
- [34] Donghyuk Kim, Jae-Young Kim, Wontak Han, Jongsoo Won, Haerang Choi, Yongkee Kwon, and Joo-Young Kim. 2024. Darwin: A DRAM-Based Multi-Level Processing-in-Memory Architecture for Column-Oriented Database. *IEEE Transactions on Emerging Topics in Computing* (2024).
- [35] Y. Kim et al. 2012. A case for exploiting subarray-level parallelism (SALP) in DRAM. *ACM SIGARCH Computer Architecture News* 40, 3 (2012), 368–379.
- [36] Y. Kim, W. Yang, and O. Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. *CAL* 15, 1 (2015), 45–49.
- [37] Y Kwon, K Vladimir, et al. 2022. System architecture and software stack for GDDR6-AiM. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE, 1–25.
- [38] Young-Cheon Kwon et al. 2021. 25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2 tflops programmable computing unit using bank-level parallelism, for machine learning applications. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 64. IEEE, 350–352.
- [39] George Kyriazis. 2012. Heterogeneous system architecture: A technical review. *AMD Fusion Developer Summit* (2012), 21.
- [40] 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.
- [41] S. Lee et al. 2021. Hardware architecture and software stack for PIM based on commercial DRAM technology: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 43–56.
- [42] M. Lenjani et al. 2020. Fulcrum: A simplified control and access mechanism toward flexible and practical in-situ accelerators. In *HPCA*. 556–569.
- [43] M. Lenjani et al. 2022. Gearbox: A case for supporting accumulation dispatching and hybrid partitioning in PIM-based accelerators. In *ISCA*. 218–230.
- [44] M. Lenjani, A. Ahmed, and K. Skadron. 2022. Pulley: An Algorithm/Hardware Co-Optimization for In-Memory Sorting. *CAL* 21, 2 (2022), 109–112.
- [45] J. Leskovec et al. 2012. Learning to discover social circles in ego networks. *Advances in neural information processing systems* 25 (2012).
- [46] S. Li et al. 2017. DRISA: A DRAM-based reconfigurable in-situ accelerator. In *MICRO*. 288–301.

- [47] S. Li et al. 2020. DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator. *CAL* 19, 2 (2020), 106–109.
- [48] S Li, D Reddy, and B Jacob. 2018. A performance & power comparison of modern high-speed dram architectures. In *Proceedings of the International Symposium on Memory Systems*. 341–353.
- [49] M. Marazzi, T. Sachsenweger, F. Solt, P. Zeng, K. Takashi, M. Yarema, and K. Razavi. 2024. HiFi-DRAM: Enabling High-fidelity DRAM Research by Uncovering Sense Amplifiers with IC Imaging. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*.
- [50] H. Martinez et al. 2026. In-memory operators for medical image processing. *Future Generation Computer Systems* 174 (2026), 107939.
- [51] Micron. 2017. TN-40-07: Calculating Memory Power for DDR4 SDRAM. (2017).
- [52] Micron. 2025. Integrating and Operating HBM Memory. <https://assets.micron.com/adobe/assets/urn:aaid:aem:275edf31-79e3-4b6c-8bbd-a233babe9281/original/as/micron-hbm2e-memory-wp.pdf>.
- [53] G. E. Moore. 1998. Cramming more components onto integrated circuits. *Proceedings of the IEEE (Proc. IEEE)* 86, 1 (1998), 82–85.
- [54] F. Murtagh and P. Contreras. 2012. Algorithms for hierarchical clustering: an overview. *Wiley Interdisciplinary Reviews (WIREs): Data Mining and Knowledge Discovery* 2, 1 (2012), 86–97.
- [55] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungrinun. 2022. A modern primer on processing in memory. In *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*. Springer, 171–243.
- [56] NVIDIA Corporation. 2020. NVIDIA A100 Tensor Core GPU. <https://www.nvidia.com/en-us/data-center/a100/>. Accessed: 2024-06-02.
- [57] Ataberk Olgun et al. 2021. QUAC-TRNG: High-throughput true random number generation using quadruple row activation in commodity DRAM chips. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 944–957.
- [58] Joaquín Olivares, Javier Hormigo, Julio Villalba, and Ignacio Benavides. 2004. Minimum Sum of Absolute Differences implementation in a single FPGA device. In *International Conference on Field Programmable Logic and Applications*. Springer, 986–990.
- [59] G. F. Oliveira, A. Olgun, A. G. Yağlıkçı, F. Bostancı, J. Gómez-Luna, S. Ghose, and O. Mutlu. 2024. MIMDRAM: An End-to-End Processing-Using-DRAM System for High-Throughput, Energy-Efficient and Programmer-Transparent Multiple-Instruction Multiple-Data Processing. In *HPCA*. 186–203.
- [60] D. Patterson et al. 1997. A Case for Intelligent RAM: IRAM. *IEEE Micro* 17, 2 (Mar./Apr. 1997), 34–44.
- [61] Samsung. 2025. PIMSimulator. <https://github.com/SAITPublic/PIMSimulator>. Accessed June 4, 2025.
- [62] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *MICRO*. 273–287.
- [63] F. A. Siddique et al. 2024. Architectural Modeling and Benchmarking for Digital DRAM PIM. In *IISWC*. IEEE, 247–261.
- [64] K. Simonyan and A. Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
- [65] SK hynix Inc. 2023. AXDIMM: Acceleration DIMM Specification. <https://www.skhynix.com/>.
- [66] H. Son et al. 2025. PIMnet: A Domain-Specific Network for Efficient Collective Communication in Scalable PIM. In *HPCA*. IEEE, 1557–1572.
- [67] H. S. Stone. 1970. A logic-in-memory computer. *IEEE Transactions on Computers (TC)* 100, 1 (1970), 73–78.
- [68] W. Sun et al. 2021. ABC-DIMM: Alleviating the bottleneck of communication in DIMM-based near-memory processing with inter-DIMM broadcast. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 237–250.
- [69] TechPowerUp. 2025. AMD EPYC Processors. https://www.techpowerup.com/cpu-specs/?f=generation_AMD+EPYC. Accessed June 4, 2025.
- [70] Boyu Tian, Yiwei Li, Li Jiang, Shuangyu Cai, and Mingyu Gao. 2024. Ndpbridge: Enabling cross-bank coordination in near-dram-bank processing architectures. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 628–643.
- [71] T. Tracy, Y. Fu, I. Roy, E. Jonas, and P. Glendenning. 2016. Towards machine learning on the automata processor. In *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*. Springer, 200–218.
- [72] X. Wang, J. Yang, Y. Zhao, X. Jia, R. Yin, X. Chen, G. Qu, and W. Zhao. 2022. Triangle Counting Accelerations: From Algorithm to In-Memory Computing Architecture. *IEEE Trans. Comput.* 71, 10 (Oct 2022), 2462–2472. <https://doi.org/10.1109/TC.2021.3131049>
- [73] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 1–12.
- [74] Wikipedia. 2025. Nvidia GPUs. https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units. Accessed June 4, 2025.
- [75] S. Williams et al. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.
- [76] L. Wu et al. 2021. Sieve: Scalable in-situ DRAM-based accelerator designs for massively parallel k-mer matching. In *ISCA*. 251–264.
- [77] Lingxi Wu et al. 2022. DRAM-CAM: General-purpose bit-serial exact pattern matching. *IEEE Computer Architecture Letters* 21, 2 (2022), 89–92.
- [78] W A Wulf et al. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.
- [79] T Xie, Z Zhu, et al. 2025. UniNDP: A Unified Compilation and Simulation Tool for Near DRAM Processing Architectures. In *HPCA*. IEEE, 624–640.
- [80] S. Xu, X. Chen, Y. Wang, Y. Han, X. Qian, and X. Li. 2018. PIMSim: A flexible and detailed processing-in-memory simulator. *CAL* 18, 1 (2018), 6–9.
- [81] Y. Zhou and others. 2005. SiO2 Matrix from PARSEC Collection in SuiteSparse Matrix Collection. <https://sparse.tamu.edu/PARSEC/SiO2>.
- [82] R. M. Yoo, A. Romano, and C. Kozyrakis. 2009. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *IISWC*. 198–207.
- [83] C. Yu, S. Liu, and S. Khan. 2021. Multipim: A detailed and configurable multi-stack processing-in-memory simulator. *CAL* 20, 1 (2021), 54–57.
- [84] Zhe Zhou, Cong Li, Fan Yang, and Guangyu Sun. 2023. Dimm-link: Enabling efficient inter-dimm communication for near-memory processing. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 302–316.